RESEARCH INSIGHTS

Modern Security Vulnerability Discovery

NCC Group

CONTENTS

AUTHOR

NCC Group

This paper is the combined effort of numerous individuals within NCC Group across various disciplines and offices.

- Aaron Adams
- Pete Beck
- Jeremy Boone
- Zsolt Imre
- Greg Jenkins
- Edward Torkington
- Ollie Whitehouse
- Peter Winter-Smith
- David Wood

INTRODUCTION

The identification of vulnerabilities and understanding what is involved in their exploitation has numerous applications in both the attack and defence side of cyber security. The way in which software vulnerabilities are discovered has evolved considerably over the last 20 years in terms of techniques, efficiency and the complexity of the issues found. What was once a mostly manual process has, over time, become increasingly automated and augmented to facilitate not only discovery, but also triage and sometimes exploitation.

However, while this automation and augmentation has helped the process of vulnerability discovery considerably, it has not addressed all the challenges faced with increasingly esoteric[1] weaknesses being discovered by highly skilled individuals. These often subtle logic bugs are challenging to find in an automated fashion and typically rely on a large body of knowledge and experience being applied to a particular set of circumstances.

This paper is intended for individuals with a technical background who are responsible for identifying, understanding, mitigating or responding to security vulnerabilities in software. The paper is technical in nature, although high level, and is intended to provide a view on modern vulnerability discovery approaches in 2016.

[1] https://en.wikipedia.org/wiki/Row_hammer

Manual analysis

Introduction

Historically, manual programme analysis was the primary method researchers would use to look for flaws in software, and it is still a very popular approach. This involves manually reviewing source code or reverse engineering binaries in order to find vulnerabilities "by hand". This will often include using small scripts or tools to aid in finding some common flaws or questionable function calls, but won't typically involve automated testing or intelligent tools that can actually reason about code constructs. Most people choosing to review software manually will focus more on understanding the software and most specifically, the high-risk entry points of any identified attack surface.

The primary objective of a manual programme analysis is either to quickly home in on software locations with a high probability of vulnerability, or to take a more holistic approach to gain a thorough understanding of the target. The approach taken, as well as the resulting bug quality, will largely depend on the skill of the reviewer, as well as their familiarity with the software itself, the internals of the language used to write the programme and the operating system on which it is run.

Attack surface determination

Generally, the first thing to do when reviewing code is to determine the probable attack surface of the target before even starting to look at any source code or disassembly. Knowing the parts of a programme that are attackable will give a natural starting point for analysing code directly in a top-down approach or will serve as a guiding light while performing a more thorough code analysis using a bottom-up approach.

Most software will have some obvious attack surface: the kernel can be attacked by user-land processes issuing system calls; an HTTP server can be attacked by a client sending HTTP requests; a setuid binary can be attacked by manipulating environment variables or command-line arguments. A lot of software might also include less obvious attack surfaces that only thorough understanding and analysis will uncover. An interesting historical example of a non-obvious attack surface would be the discovery of "shatter" attacks[2] against privileged applications running on Windows. This type of attack would be very difficult to discover for a researcher who did not have an in-depth understanding of Windows. Focusing on a target programme in isolation would make them blind to these additional components of the attack surface.

Researchers approach attack surface modelling in different ways. Some simply keep notes, others keep internal mental models, and others will use graphical software that allow different endpoints to be plotted to visually show a representation of the system being targeted. The amount of time spent fleshing out the attack surface will depend on the time being committed to a project, or just the style and approach of the researcher. The attack surface model will, however, constantly be revisited and reworked as a more in-depth understanding is developed.

Analysis approaches

There are effectively two common approaches to manual programme analysis, dubbed "top-down" and "bottom-up" analysis. Realistically things aren't so black and white though. Most commonly a combination of both approaches described will be used, as both offer some advantages and can help understand things at different phases of a review. These two approaches are summarised below. An exhaustive book on comprehensive manual programme analysis is The Art of Software Security Assessment (often referred to as TAOSSA)[3], which is highly recommended for anyone planning on pursuing manual bug finding.

[2] https://en.wikipedia.org/wiki/Shatter_attack
[3] https://www.amazon.co.uk/Art-Software-Security-Assessment-Vulnerabilities/dp/0321444426

Top-down analysis

This is probably the most common approach to finding bugs in a programme. A researcher will make an informed decision, based on an attack surface analysis, about where to start looking for bugs. They will then often search for some potentially dangerous function calls or integer operations in code related to that attack surface and try to work backwards to determine if they can actually be abused.

Researchers will begin reviewing code and understanding the programme only from this "top" part of functionality and then work down into the deeper portions of code. Often, this allows them to disregard how the rest of the programme works or is initialised. This can lead to quick results, but, depending on the complexity of the programme, can also leave the researcher blind to various configuration-specific quirks.

Despite the top-down approach being intended to target interesting functionality, it unfortunately can still end up leading to numerous false positives. Consider a case where a programme uses the notorious strcpy function. It's actually entirely possible to use this function safely, but a manual reviewer doing a top-down analysis might search for uses of this function as possible entry points to more closely focus their analysis. In the event that the programme's use of this function is ubiquitous, but also happens to be largely benign, the researcher could be presented with hundreds of thousands of results, with no clear indication of where to usefully start their top-down analysis from or with countless dead-ends when they do finally start.

There are still times where searching for particular functions can be extremely useful though and the example above isn't always the way things pan out. Consider a case where you've identified some internal function that is relatively infrequently used, but has a high probability of being used incorrectly. An example might be a function whose return value was not consistently checked, and could be easily influenced to return an unexpected error. Searching for all instances of this function might lead to a much more acceptable signal to noise ratio than a more common library function like strcpy.

In reality, the approach used for top-down analysis will often be determined on-the-fly and based on the complexity of the programme and the results of initial probing. The fruitlessness of certain approaches will often quickly become apparent after the initial "sniffing around" phase.

A realistic example of using the top-down approach against an HTTP server would be starting with the realisation that HTTP servers almost always have a significant amount of code related to configuration and plugin management, which is far less interesting from a security perspective than the code handling which handles requests over the network. After identifying code responsible for handling HTTP requests, a researcher may simply look for questionable function calls or code constructs (like integer operations) in the source or disassembly that is associated specifically with that functionality. Once these calls are found, investigations can be performed to see if and how attacker-provided input might be provided to the questionable calls through the HTTP headers or other request data.

Bottom-up analysis

This approach is much simpler to understand, but can be much more time consuming and complicated in practice. It will often lead to the discovery of significantly more vulnerabilities than the top-down approach. Moreover, it can lead to the discovery of the type of complex or intricate bug that can only be found through manual rather than automated analysis. The idea of bottom-up analysis is to effectively start from a given point in the programme, typically the main function, and slowly understand and analyse all functionality. Realistically, due to time constraints, this will involve certain areas that have nothing to do with any attack surface being skipped. Using this approach allows one to slowly build up a thorough understanding of all of a programme's components, custom functions, and so on. As a result, it becomes possible to start identifying new attack surfaces and application-specific vulnerability constructs that would be missed by simply looking for common mistakes.

Using the bottom-up approach typically involves keeping significantly more documentation and is often the best approach when a researcher will be spending lots of time on a project. Even if time constraints prevent all code from being analysed, it will help to highlight the areas to focus on in future engagements.

For interesting examples of the type of vulnerabilities that can be discovered from a long-term bottom-up analysis see the Windows operating system flaws discovered by James Forshaw of Google's Project Zero[4] .

Functionality-specific analysis

Functionality-specific analysis can also be useful when the sheer size of a codebase makes it impossible to do a full bottom-up analysis. For example, an operating system kernel is so massive that it's not feasible to review or understand all of the functionality. However, following a reasonably thorough bottom-up analysis, one can do future engagements that consist of smaller functionality-specific reviews that focus on certain bug classes or certain interactions, and effectively ignore other functionality that's not of interest. In this way, a researcher can slowly build out a more thorough understanding of the kernel, piece by piece. This might consist of re-reading the same pieces of core functionality multiple times, but with a new frame of reference, and can often allow bugs that were previously missed to be discovered thanks to a new understanding or perspective.

Relevance

Several other approaches to finding bugs are discussed later in this paper. These have become increasingly popular, and in many ways are more commonly used for vulnerability discovery than manual analysis. Many would argue that these more modern approaches are much easier to pursue. Although this is often true, for the foreseeable future there will always be a degree of shallowness to the bugs found in this way. Static analysis and fuzzing are capable of finding a lot of different bug classes, but they still lack the ability to find abstract logical issues that are usually only found with the help of the thorough and holistic understanding of a programme's behaviour that is achieved through manual analysis. Additionally, many researchers still prefer manual analysis because it helps to avoid the massive signal to noise problem that can result from automated analysis. By doing things yourself, it is possible to focus on vulnerabilities that will have a high probability of exploitation (as opposed to those which result only in a crash). A significant number of vulnerabilities being discovered on a day-to-day basis are the product of manual program analysis.

Conclusion

At an abstract level most manual programme analysis follows a simple flow that consists of only a small set of actual approaches. Until automated analysis becomes significantly more robust, there will always be room for manual programme analysis and it should still be a significant part of most assessment approaches, while being augmented with more modern automated analysis.

[4] https://code.google.com/p/google-security-research/issues/list?can=1&q=reporter%3Aforshaw

Dynamic analysis

Overview

Dynamic analysis is a term used to describe the testing or analysis of software that is performed while running the programme. It usually refers to testing that is conducted by an automated tool. It allows a researcher to observe the internal state of a running program and see how it changes in response to different inputs.

Dynamic analysis can be both passive and active. Passive dynamic analysis merely observes the program as it runs and records its actions. Active dynamic analysis changes a program's behaviour in some way; for example, causing certain actions to fail to investigate error handling mechanisms.

Dynamic analysis tools mostly fall into one of two categories. The first type of tool modifies the programme under test to add extra code to perform the desired action. This modification can be done at compile time, which has the advantage of introducing very little performance hit but requires source code, or at run time - which only requires the binary. The second type of tool runs the program within some kind of test harness. Most commonly, this would use an operating system's debugging API to allow the harness to observe and/or modify the behaviour of the programme under test. This incurs a much larger penalty in terms of performance, but is considerably more flexible and powerful.

One advantage of automated dynamic analysis is that it scales very easily. Typically, each instance of the programme under test is independent and therefore, running multiple instances simultaneously is reasonably straightforward given the necessary computing resources.

Vulnerability discovery

Most passive dynamic analysis techniques do not directly identify vulnerabilities. What they do is enable researchers to observe the state of the programme or measure the effectiveness of other techniques such as fuzzing.

Active dynamic analysis techniques can be used to find vulnerabilities directly. This could be through exercising parts of the code that are difficult to test, or through additional sanitisation and validation of runtime data.

Debuggers are essentially manual dynamic analysis tools that are used extensively by researchers. However, since the focus of this part of the paper is automated dynamic analysis, they are not discussed further.

Code instrumentation

Instrumentation is the most common form of passive dynamic analysis. It allows a researcher to observe the state of a running programme which can greatly aid vulnerability discovery.

One technique is known as API tracing and the Linux programme strace is a good example. Each time an API of interest is called a log entry is generated, which typically includes details of the caller and any parameters. For example, it can easily show if attacker-controlled input is used by insecure APIs such as strcpy. It is much easier to use a technique like this to directly observe data being passed into functions than attempting to obtain the same information by performing full dataflow analysis for a binary.

Code coverage analysis records how frequently each part of the programme is executed. It can be used to measure the effectiveness of a test set, possibly by demonstrating a test set is incomplete, or by eliminating cases that do not contain new unique code coverage. It is also used by some fuzzers to reduce the size of the test set without losing coverage.

Code coverage analysis is also used by some whole-program optimisers during compilation to group commonly executed code together in memory, which reduces the working set size for a binary. This is why some functions are split into several chunks in different parts of a binary. The code that the compiler thinks represents "normal" execution would be in one location while the rest would be stored elsewhere.

Another notable instrumentation technique that is increasing in popularity is the use of hypervisors to instrument functionality that is hard to analyse dynamically using more traditional techniques. The main research prompting this was a project called Bochspwn[5] whereby the Windows kernel was instrumented to automate the identification of kernel time-of-check time-of-use (TOCTOU) race condition vulnerabilities, to great success. This involved logging user-land memory accesses from the kernel to identify code that accessed the same memory location two or more times in the same code path. A more recent incarnation of this, called Xenpwn[6], instrumented the Xen hypervisor in order to identify similar TOCTOU issues.

Fault injection

Fault injection is an active dynamic analysis technique used to simulate failures of a program that would be unlikely to occur during normal operation. It is commonly used to test error handling for cases that are difficult to engineer without causing other failures. For example, out-of-memory conditions or open file handle limits can easily be simulated by causing memory allocation functions to fail only in one process, or when called by specific functions, using tools such as Valgrind[7].

This allows parts of the programme that would not be executed during normal operation to be tested. Bugs in error handling are far less likely to be found during development as the code in question would rarely be exercised.

Another advantage of fault injection is that it allows testing of potentially fatal system-wide issues without impacting other processes. Causing memory allocations to actually fail would likely render the testing system unusable.

Fault injection should not be confused with general Fault Injection aka Fuzzing which is discussed later in this paper.

Runtime verification

Another technique related to fault injection is runtime verification. It involves inserting additional runtime code to verify the validity of data before it is used. This is done by initialising variables to deliberately invalid values and testing for these values when a variable is used.

Tools such as Address Sanitizer[8] and Application Verifier[9] both use this technique. In combination with fuzzing it is a method of identifying vulnerabilities. Many of the bugs it finds are very hard to identify using other techniques such as static analysis or even manual code review. It can detect bugs including uninitialised pointer use, buffer overflows and use-after-free conditions. For example, the tools can detect use-after-free conditions by adding code to memory management routines to set buffers to specific values when the memory is allocated or freed. The values are chosen to be invalid addresses so that any attempt to use them as pointers will cause an exception. The tool can then determine the type of bug based on the address the code attempted to access.

[5] http://j00ru.vexillium.org/?p=1695
[6] https://www.ernw.de/download/xenpwn.pdf
[7] http://valgrind.org/
[8] https://github.com/google/sanitizers
[9] https://www.microsoft.com/en-gb/download/details.aspx?id=20028

SMT solvers

Satisfiability Modulo Theories (SMT) solvers, or constraint resolvers, attempt to find a set of values that meet a certain set of conditions. They have numerous applications in computer science, several of which are relevant to dynamic analysis.

At the conceptual level, SMT solvers allow the following kinds of question to be answered:

- How must the input change to make program execution follow the other path at this branch?
- Is there any input that can cause this calculation to overflow?
- Is there any input that causes both these function calls to execute?

Common SMT solvers used in security research include CVC4[10] , STP[11] and Z3[12] - the first two of these are academic projects, and the third is by Microsoft Research.

SMT solvers have been used in a number of dynamic analysis tools to increase their effectiveness. They are generally used with a technique called Symbolic Execution whereby a programme is analysed to determine how different inputs cause different parts of the code to execute. This is normally achieved using an interpreter and symbolic values for inputs. The result is a set of expressions and constraints in terms of input symbols for following different branches. The SMT solver can then attempt to find inputs that cause specific instructions to execute.

For example, a common fuzzing technique is to mutate the input data, while monitoring the code coverage graph to understand whether the mutation allowed new code to be reached. With an SMT solver, input that exercises specific code branches can be intelligently generated, instead of brute-forced by traditional fuzzing.

However, this added intelligence can be extremely computationally expensive. The SAGE fuzzer from Microsoft uses this technique to maximise code coverage.

A novel application of SMTs from Microsoft Research was generating a clean input that reproduced a particular crash[13]. For example, if an application crashed while editing a sensitive document, users would likely be reluctant to submit crash dump files in case they contained parts of the document. The solution was to rerun the application and record every branch taken, then use Z3 to generate an input that would follow exactly the same code path. Users could then send the generated file to the application vendor without the risk of compromising private data.

SMT solvers have also been used by a number of static analysis tools to significantly reduce the number of false positives, by allowing path sensitive analysis. This involves computing the constraints necessary to follow a specific path and determining if that is consistent with exercising the vulnerable code. Some static analysis tools refer to this capability as "value tracking".

[10] http://cvc4.cs.nyu.edu/web/
[11] http://stp.github.io/
[12] https://github.com/Z3Prover/z3
[13] http://research.microsoft.com/en-us/projects/betterbug/castro08better.pdf

Common tools

The following sections describe some of the common tools available to perform dynamic analysis. Several do not require the source code of the application being tested. AFL, Address Sanitizer and Application Verifier are standalone tools used for vulnerability finding, while Detours and Pin are frameworks to be used in building other dynamic analysis tools.

American Fuzzy Lop (AFL)

AFL[14] is a fuzzer that uses compile-time instrumentation and genetic algorithms to automatically discover new, interesting test-cases that trigger new internal states of the target binary. It is currently one of the most popular fuzzers and has found numerous vulnerabilities in a wide variety of applications. While not purely a dynamic analysis tool, its extensive use of dynamic analysis makes its inclusion in this section appropriate.

Although AFL performs best when applications can be recompiled to add its instrumentation, a binary-only mode is also available. AFL also contains lots of interesting features for security researchers.

- Given one test case, try to generate a smaller one that follows the same code path.
- Generate valid files for a given parser without a reference set.
- Infer the format for input files by observing code flow in the parser.

AFL is still under active development and the available features are likely to increase.

Address Sanitizer

Address Sanitizer (ASan) is a fast memory error detector. It comprises a compiler instrumentation module and a runtime library. It can detect various memory-related issues including:

- Out-of-bounds accesses to heap, stack and globals.
- Use-after-free.
- Use-after-return (to some extent).
- Double-free, invalid free.

Address Sanitizer is implemented in both the Clang and GCC compilers. In order for ASan to operate, the target binaries must be recompiled with the appropriate command line flags. Therefore, access to the target source code is mandatory.

ASan is currently used extensively by the developers of the Firefox and Chrome web browsers and has found numerous memory corruption vulnerabilities. It is regularly combined with fuzzing tools such as AFL, as this greatly improves the quality of the results.

Application verifier

Application Verifier is a runtime verification tool developed by Microsoft. It can find subtle programming errors that can be extremely difficult to identify with normal application testing. Application Verifier operates by simulating various error conditions to understand how the application handles these rare edge cases. Consequently, it can identify errors related to heap corruption, invalid handles and critical section usage.

The most effective method of finding vulnerabilities with Application Verifier is to enable its various checks then run the application under a debugger. This means that any issue found will cause the debugger to break.

[14] http://lcamtuf.coredump.cx/afl/

Detours

Detours[15] is a library written by Microsoft Research that allows dynamic instrumentation of binaries. It enables a researcher to build applications to hook functions at runtime in order to perform dynamic analysis. Despite the description as an instrumentation tool, Detours can be used to write tools that perform either passive or active analysis. Detours does not require the source code of the programme under analysis, however, debugging symbols enable some additional functionality such as looking up function addresses at runtime.

The Detours library enables interception of function calls. It achieves this by replacing the first few instructions of a target function with a jump to a user supplied detour function. The overwritten instructions are preserved in a trampoline function which comprises the overwritten instructions and a jump to the remainder of the code. The code for the trampoline function is created automatically by the Detours library. All a user needs to provide is a pointer that gets set to the appropriate code once a detour has been put in place.

Microsoft makes extensive use of Detours in its own testing. In fact, the Microsoft compilers were modified to make function hooking in the emitted binaries significantly easier. Every function has a fixed pattern at its start and end to enable safe and efficient detouring. Specifically, the first two bytes of every function are a NOP-equivalent, while the five bytes before each function are all NOPs. These can be safely modified by Detours to hook calls to a given function.

The free version of Detours only supports Windows on 32-bit x86 platforms. The Microsoft-internal and professional versions also support x64 and ARM CPUs.

Pin

Pin[16] is a framework written by Intel that allows dynamic instrumentation of binaries at runtime. Like Detours, it does not require the source code of binaries under test. It can be used to develop tools that perform both passive and active dynamic analysis. For example, it is very common for Pin to be used alongside a fuzzer to measure code coverage.

Pin can perform instrumentation at a very low level – down to individual instructions if required. It can also add instrumentation at basic block or function level. It also supports multiple operating systems, including Windows, Linux, OSX and Android on both 32-bit and 64-bit Intel CPUs.

Conclusion

Dynamic analysis covers a wide range of programme instrumentation methods. Passive dynamic analysis can significantly increase the efficiency of other vulnerability discovery techniques such as fuzzing. Active dynamic analysis can be used to discover vulnerabilities that are extremely difficult to detect by any other method.

There are bugs that dynamic analysis would be very unlikely to ever discover. The same can be said for static analysis and manual code review. However, the best method for finding vulnerabilities is always likely to be a combination of techniques. AFL has shown that by combining relatively simple fuzzing with dynamic analysis, the efficiency of fuzzing can be massively increased, resulting in many vulnerabilities being found.

The frameworks available with which to build bespoke dynamic analysis tools are incredibly flexible and powerful. The only limits on what can be achieved are available computing resources and imagination.

[15] http://research.microsoft.com/en-us/projects/detours/
[16] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

Static code analysis

Overview

Static analysis and Static Application Security Testing (SAST) are terms used to describe the testing or analysis of software that is performed without needing to actually execute the programme. The term is usually applied to testing that is conducted by an automated tool, whereas human analysis is typically called security-focused code review or simply manual analysis.

The vast majority of static analysis tools will perform their analysis on a programme's source code, although some tools do support the analysis of object, byte or even machine code. These days, you can find a static analysis tool for pretty much every programming language, with varying degrees of usefulness.

The primary objective of a static analysis tool is to gain an understanding of the target software's behaviour, usually with the aim of uncovering security, privacy and quality issues. As such, static code analysis tools can be extremely valuable when performing vulnerability research.

The importance of intelligent programme analysis

Many static analysis tools do not attempt to deeply understand the behaviour of the software being tested, instead opting to simply scratch the surface with naïve grep-like pattern matching. This is unfortunate, because it can increase the rate of false positive detections. Separating the wheat from the chaff can place significant triage burden on the vulnerability researcher.

For example, a naïve tool might simply say: "Here are all the places that strcpy is called", which is to some extent helpful, but still places considerable onus on the researcher to verify which instances are potentially vulnerable. In contrast, an intelligent tool would say: "Here are the places where external input to the programme could induce a buffer overflow in strcpy".

These types of naïve tools should be avoided. They can produce useful results, but these tend to be buried deep inside a haystack of false positives. Use these tools only if you have an excess of free time and suspect the target software contains low hanging vulnerabilities.

So then, what constitutes intelligent programme analysis? One could write hundreds of pages on this topic, but we will try to summarise it in simple terms here. Intelligent static analysis tools will frequently take additional steps beyond simply tokenizing and parsing the code into an abstract syntax tree. Generally, a smart static analyser will generate an intra-procedural, or better, an inter-procedural data-flow graph. This construct indicates how data will propagate through the programme via function calls and assignment operations. For static analysis of C/C++ code, an excellent tool that has these capabilities is Joern[17].

Data-flow analysis is often supported by some form of abstract interpretation and value tracking. These are techniques for evaluating mathematical and boolean expressions in an attempt to characterise the behaviour of the program. For example, this is useful for automatically determining whether user-supplied input can control the evaluation of a series of conditions or expressions that are needed to induce a vulnerability found deep within the code.

But how does a static analysis engine understand the concept of an array, and that it should never be accessed out of bounds? And for that matter, how does the static analyser understand the semantics of security-impacting functions such as the previously-mentioned strcpy? These answers are described in the next section.

[17] https://github.com/octopus-platform/joern

Rulesets

Essentially, a rule is a set of conditions that, when met, instruct the static analysis tool to trigger a vulnerability warning. In the above example, a static analyser might have a rules which explains that an array can be imagined as a sequence of bytes in memory, and that the [] operators access the array at a certain offset, and this offset must never exceed the size of the array. Static analysers will have hundreds of other such rules. For example, a ruleset might encode the notion that a pointer must never be re-used after it has been passed to free, or that when calling strcpy, the length of the source buffer must never exceed the length of the destination buffer.

At this point, it should be clear that in order to detect a diverse and comprehensive set of security vulnerabilities, the static analyser must have a deep understanding of not only the programming language syntax, but also the compiler, target architecture, and the semantics of security-impacting functions in the standard library and common frameworks. Each of these can introduce architectural abstractions that can confuse a naïve tool, but more importantly, they may encourage programming patterns that can have unsafe security consequences. Therefore, the static analyser must understand concepts such as dynamic memory allocation, must be able to resolve layers of abstraction, and must be able to follow method calls made via function pointers or dynamic dispatch techniques. If it doesn't do these things, it will fail to detect many genuine vulnerabilities.

For example, the static analysis tool must know whether the target is 32-bits or 64-bits, as this can inform the analysis engine of when an integer type will overflow. Similarly, a static analysis tool that doesn't understand the semantics of abort (programme execution ends here), malloc (dynamic memory is created here), or copy_from_user (untrusted user-supplied data enters the system here) is going to yield poor false positive and false negative rates. A static analysis tool must understand these concepts, and many others, in order to thoroughly analyse software.

This is often the difference between commercial and open source static analysis tools. Commercial static code analysis tools will be packaged with a diverse collection of rulesets and support analysis techniques such as data-flow graph traversal and abstract interpretation. On the other hand, free static analysis tools tend to be a mixed bag. Some come with a handful of rules, while others ship with zero rulesets, expecting you to write you own. But this is not necessarily a bad thing. In fact, we should celebrate static analysis tools that expose a flexible detection engine to the user. This is where we, as security researchers, can leverage our expertise.

Flexible detection engines allow us to extend the analyser to detect entirely new classes of vulnerabilities. Perhaps you are performing a security assessment of a software system that is unique in how it allows untrusted data to enter the system, such as a custom my_recv implementation or a hypothetical foo_protocol_packet_parser function. Or perhaps you've stumbled across a new coding pattern that describes a novel class of vulnerability that was previously unknown. This is where the fun starts.

Modeling untrusted data entry points

When conducting a security assessment of a new software product, one of the first and most important steps will be to enumerate the attack surface. Typically, this involves building a comprehensive understanding of how data enters and exits the system at its boundaries. When data first enters a software system, we must treat this data as untrusted, and it should therefore be validated and sanitised before being used.

In static code analysis tools that utilise data-flow analysis, a data entry point is often called a source, whereas the location that consumes this untrusted data is often called a sink. An intelligent static analysis engine will allow its users to write custom models that describe the semantics of additional sources and sinks. Doing so will enable the analysis engine to track this data as it propagates through the program. The ultimate goal is to ensure that untrusted data is sanitised before transitively tainting other data, or being used in a sensitive operation, such as controlling the size argument to memcpy.

As mentioned previously, custom modelling will come into play when auditing software that diverges from established coding patterns. In these cases, a static analysis tool simply cannot be expected to infer the input and output semantics of the software. For example, when you encounter a custom library, you should strongly consider modelling its API, ensuring that the static analysis engine can track how the untrusted data flows between the library and the application.

Automation and variation hunting

When we perform security-focused code review, we sometimes discover unique coding patterns that lead to original security vulnerabilities. These emergent classes of vulnerabilities could be unique to an individual developer's particular coding style, or they may be unintentionally encouraged by a poorly-designed API within a library or framework.

The ambitious security researcher will want to understand whether these bad smells are repeated elsewhere in the target code base. Conducting this search by hand can be tedious and time consuming, so this type of variation hunting is simply begging to be automated through the use of a static code analysis tool.

For example, in recent memory, there was a rush by commercial Static Application Security Testing vendors to demonstrate they could detect the general patterns exhibited by the 'Heartbleed'[18] [19] and 'Goto Fail'[20] [21] vulnerabilities. In some cases, the commercial tools did not detect the vulnerability, and new detection heuristics or rules had to be created.

This reveals how a static analysis engine can be tricked by even the most subtle variation of a seemingly well understood (yet flawed) coding pattern. But don't be dismayed! This is actually great news for a security researcher. If you ever stumble across what appears to be an interesting pattern that leads to a vulnerability, consider creating a ruleset in a flexible tool like Joern[22]. You're likely to find variations elsewhere in the code base.

Conclusion

In many cases you do not need to bother writing custom models or rule sets if dealing with an application that uses common languages and frameworks. In fact, finding vulnerabilities using static analysis can often be as simple as running an off-the-shelf tool in the default configuration. Although, if the code is open source it is likely to have already been scanned by every static analysis tool on the planet, multiple times over. Just take a look at the Coverity Scan project[23], which has scanned thousands of open source code bases. But if you're looking at a fresh code base that has never been exposed to static analysis tools, get ready and prepare yourself for a bonanza of vulnerabilities.

[18] http://blog.regehr.org/archives/1125
[19] http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html
[20] http://security.coverity.com/blog/2014/Feb/a-quick-post-on-apple-security-55471-aka-gotofail.html
[21] http://blog.klocwork.com/static-analysis/rather-than-fail-goto-success/
[22] https://github.com/fabsx00/joern
[23] https://scan.coverity.com/projects

Static software composition analysis.

Overview

When analysing software, it is often necessary to understand how it is built and what third party components it depends upon. This type of software analysis is becoming increasingly important as we look to identify the weak links and deal with monolithic blobs such as firmware for embedded systems. The approaches used for composition analysis can be applied universally across a variety of platforms such as mobile, web application, desktop or embedded.

Being able to understand what software is comprised of can not only facilitate weak link identification but also provide indicators as to means of exploitation as well as general security posture.

In the sections which follow we summarise the approaches used to identify how software is composed.

Static Open Source Enumeration & analysis

An important element of software maintenance involves the software bill-of-materials (BOM), but maintaining such a BOM by hand can be cumbersome or impossible due to the complex nature of dependency chaining. For example, your software may depend upon a third party package that, unbeknownst to you, depends upon additional third party packages, through either static or dynamic linkage.

Consequently, it can be quite difficult to understand which versions of what third-party components are built into your application. And yet, being able to quickly answer this type of question is an essential aspect of a mature Security Incident Response process, such as when a newly patched vulnerability affecting specific versions of an open source library is disclosed, and it is necessary to determine whether you need to update your software.

A number of static analysis techniques and approaches can be employed to solve the third party library identification and versioning problem. Commercial tools such as BlackDuck, Protecode, TripleCheck or Palamida will parse code or compiled binaries and apply a set of heuristics or rules that determine which libraries are compiled in. This information may be derived directly from proprietary databases or from package manager metadata included with the source repository.

After library and version information has been obtained it can then be cross-referenced with vulnerability databases such as NIST NVD in order to identify those libraries that represent a security risk.

Static closed source enumeration and analysis

At a very high level, the aspects of software composition which could usefully be analysed by tools are listed below.

- Platform binary
- CPU architectures
- Compiler and versions
- Third party external libraries used, and their versions
- Compiler level protections leveraged
- Operating system protections leveraged

Examples of tools which facilitate this include checksec[24] and WinBinaryAudit[25].

Beyond these we are increasingly seeing automation to a far greater depth of analysis, at the level of both individual binaries and monolithic firmware blobs. These enhanced capabilities include enumeration and at times automatic extraction around a number of other areas, such as:

- Structure
- Compression, packing/obfuscation and encryption
- Statically linked libraries and their version or if specifically vulnerable

Examples of tools that provide such functionality include binwalk[26] and fiminator[27].

Conclusion

Modern software is increasingly made up of a collection of third party libraries and components. The ability to efficiently and effectively identify the composition of software is increasingly important in vulnerability discovery as well as understanding how software may be attacked in the future.

[24] http://www.trapkit.de/tools/checksec.html
[25] https://github.com/olliencc/WinBinaryAudit
[26] http://binwalk.org/
[27] https://github.com/misterch0c/firminator_backend

Fuzzing

Overview

Fuzzing is a vulnerability discovery technique in which large quantities of data, packaged as individual 'test cases', are submitted as input to the target system – software or hardware – with the aim of causing crashes, faults or other behaviours. Typically this process involves the use of a purpose-built piece of hardware or software – a 'fuzzer' – to generate and submit the test cases. Security researchers may choose to use a publicly available fuzzing tool, or to create their own as needs dictate.

The principle behind fuzzing is to leverage the power of the computer, generally favouring quantity over quality. In an ideal setup, a researcher might hope for their fuzzer to evaluate thousands or even millions of test cases per second. Once a fuzzer has successfully induced a crash, it will typically flag the corresponding test case as interesting and continue searching for other faults or anomalous behaviour. The researcher will eventually want to manually analyse the test cases marked as interesting with a view to reproducing the fault, determining if it is likely to be exploitable and ultimately refining a working proof-of-concept exploit.

The rate at which most systems can be fuzzed in this way draws attention to a trade-off between development time and execution time of the fuzzer: sometimes at the most simple send random bits or bytes to the target will be sufficient to generate crashes, whereas a more hardened system would require a degree of configuration and tailoring such that the test cases resemble valid data. The ideal test cases will be 'correct enough' to pass early validation logic, with deliberate faults further into the structure designed to trip up the more sensitive code paths. This balancing act is at the crux of productive fuzzing, and the techniques and technology are constantly evolving.

This 'shotgun' approach to vulnerability discovery is very different from the manual and informed strategies such as code review and static analysis, and unsurprisingly it tends to yield a different flavour of bugs. For this reason, researchers will use their judgement to decide upon the best approach, often using fuzzing in combination with other techniques.

Fuzzing in the context of software vulnerability research has existed for two decades or more, and so the vulnerability landscape is also constantly changing. High-value targets which were once rich with exploitable vulnerabilities have now been fuzzed extensively over the years and most of the low-hanging fruit is gone. This diminishing supply of vulnerabilities continues to push the boundaries of what is considered to be a 'good' fuzzer and while the number of bugs being found through fuzzing may have tapered off in recent years, the ever-increasing complexity of these bugs is testament to the quality of modern tools.

Fuzzing is one of the most common vulnerability discovery techniques with valid input to a programme (say an image or document) mutated in some way then loaded into the programme, in the hope that the malformed input will be handled incorrectly causing a crash.

As previously mentioned there are various strategies when fuzzing. Some maintain a "keep it simple" approach. Charlie Miller's famous "5-lines of Python"[28] that mutates random bytes with no knowledge of the underlying structure of the data being mutated is probably the most extreme example. This is known as "dumb fuzzing". At the other end of the spectrum there are highly complex fuzzers that might use knowledge of the underlying format of the data they are fuzzing e.g. PE COFF files[29].

[28] https://fuzzinginfo.files.wordpress.com/2012/05/cmiller-csw-2010.pdf and https://www.youtube.com/watch?v=Xnwodi2CBws
[29] https://ece.uwaterloo.ca/~vganesh/Publications_files/vg2006-EXE-CCS.pdf

Others use constraint resolvers to identify which parts of the input to change to exercise different parts of the program; Microsoft's SAGE is probably the best known example.

Interestingly, empirical evidence suggests[30] that all well-written fuzzers find roughly the same number of bugs in a given time period on the same hardware. The basic dumb fuzzer requires almost no time to generate test cases, though each case has a relatively low probability of finding a bug. The intelligent fuzzer requires much more time to generate each test case so the total number it can try is much lower, even though each case has a much higher probability of finding a bug.

The most popular strategy currently combines dumb fuzzing with code coverage analysis. This allow rapid generation of test cases but provides feedback on which ones exercised new code and are therefore worth keeping as the basis for further mutations. AFL is the most popular example of this.

Test Case Management

Code coverage as a metric

In recent years, the measure of a fuzzer's quality has moved away from simply 'how many bugs it finds' and more towards achieving good code coverage. To explain this by way of a question: if your fuzzer finds no bugs, is that because the fuzzer isn't very effective, or because the software is well secured? Probably a bit of both, but certainly one can be confident that a fuzzer is working if the volume of code being exercised within the target application during execution of test cases is high and continuing to grow.

So how to ensure that a fuzzer is generating test cases that exercises as many code paths as possible? There are two main approaches: mutation-based and grammar-based fuzzing.

Mutation-based fuzzing

A mutation-based fuzzer will take known-good input and perturb it slightly – perhaps flip a random bit, or replace a randomly chosen 32-bit section of the file with a boundary value. This new mutated data will become another test case for submission to the target application. But of course, when you consider the size of a typical test case (say, for example a JPEG file) and the number of ways in which it can be mutated, the number of permutations can quickly become unmanageable. To test every possible mutation of all but the most trivial test cases is infeasible, so a fuzzer will need to make its choices wisely. Furthermore, if all test cases are based on a single known-good base case, it is likely that they will all result in execution of similar code paths, and hence be less likely to find new bugs. As an extreme example, consider an audio player being fuzzed through mutation of a single MP3 file. One might hope to find some bugs in the MP3 decoder, but it would take more than a stroke of luck to stumble upon a bug within the FLAC-parsing code. For this reason, it is important for mutation-based fuzzers to be primed with a broad set of base cases. Ideally these will span the range of functionality supported by the target and together obtain a good degree of code coverage.
Finally, for a mutation-based fuzzer to be effective it should have some knowledge of how the data it is mutating will be processed. The algorithms used by one target to parse XML data will be radically different from that of another which handles ZIP files. Correspondingly the kind of bugs in these two targets will be very different and the fuzzing techniques required to trigger bugs in one format will be ineffective for another. Furthermore, the presence of integrity checksums within file formats can often present a major stumbling block for uninformed fuzzing engines. Therefore, researchers will often create dedicated logic or rules within their fuzzer (or its configuration) to tailor it to what is known about the way that the target processes its data. Although laborious, this can yield significant improvements during fuzzing by allowing deeper and less thoroughly tested code paths to be reached.

[30] https://blogs.technet.microsoft.com/srd/2010/02/24/using-code-coverage-to-improve-fuzzing-results/

Grammar-based fuzzing

Turning this idea on its head, we arrive at grammar-based fuzzers. In this approach, rather than mutating a set of known good base cases, we feed the fuzzer with nothing other than a semantic description of what kind of data the target is designed to handle – in the form of a grammar or schema. The fuzzer will use this to synthesise new test cases from scratch, taking opportunities to strategically perturb the structure or content where appropriate.

This grammar-based approach of test case generation tends to yield a much greater signal-to-noise ratio than mutation-based fuzzing, and its capabilities are bound only by the level of detail within the grammar. However, it is not without its limitations, and in many real-world cases the data formats being dealt with simply cannot be neatly described within the language of the fuzzer. Often the language semantics are too simple to describe a rich data format, or are so complex that even defining a simple grammar is arduous and error-prone. The development time required to author a useful grammar-based fuzzer also tends to be significantly greater than that for a mutation-based fuzzer. These two approaches to test case generation are not mutually exclusive and many fuzzers use a combination of both to good effect.

Generational fuzzing

When a fuzzer finds a bug, it is often probable that other defects will exist in nearby code paths, in terms of the target's logic. For this reason, fuzzers will often want to focus their mutation efforts upon test cases similar to those that produce known crashes. Taking this a step further, there is mileage in concentrating on those test cases that simply exercise a previously unseen code path, even if no crash occurred. By feeding these 'interesting' test cases back into the pool of base cases, a mutation-based fuzzer can iteratively discover new code paths in the target, and hence more bugs.

When supplied with sufficient information about the execution state of the target, this technique can be used with profound results as demonstrated by American Fuzzy Lop[31] (AFL). Using source code instrumentation to provide the fuzzing engine with detailed information of the execution paths taken by the target as it runs, AFL demonstrates that a relatively simple mutation-based fuzzer is able to infer the structure of many complex file formats given no prior knowledge of how they should look. By initialising the same tool with a meaningless base-case and pointing it at two very different targets – say a GIF renderer or an ELF parser – it will produce increasingly intricate test cases including either valid GIFs or valid ELFs according to how the target responds to the mutations is generates.

It does this by pivoting upon test cases that exercise previously unseen code paths, constructing a tree of test cases which attempts to mirror the structure of the parsing code. Naturally, this level of code coverage will yield many novel defects, and AFL has a correspondingly impressive track record.
The tool in its original form, however, is limited to fuzzing of targets with source code available as it relies on compile-time instrumentation of the code to operate. Similar techniques can be applied using dynamic instrumentation (for example, using PIN[32] or DynamoRIO[33]) and virtualised execution, and research in this field continues.

Base case pruning

Put simply, the primary problem with mutation-based fuzzers is that such tools spend too much time executing test cases which do not result in unique crashes. This is a compound problem, impacted by many factors including the mutation algorithms, inability to detect early validation failures, and execution of redundant code paths. This latter point can be addressed by careful reduction of redundant base cases. The presence of multiple base cases which exercise overlapping code paths tends to cause production of needless test cases by the fuzzing engine, wasting valuable fuzzing time. This is a particular problem for generational fuzzers, whose set of base cases is constantly changing.

Given some knowledge of the execution path taken by the target when processing a given base case, a fuzzing engine

[31] http://lcamtuf.coredump.cx/afl/
[32] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[33] http://www.dynamorio.org/

can identify the presence of unnecessary samples within its set of base cases: if all of the code reached during execution of a given base case is also reached when executing some combination of others in the pool, then it is unlikely that mutation of this base case will yield any unique crashes that wouldn't be found otherwise. By iteratively applying this as a pruning algorithm, a fuzzer can strive to eliminate unnecessary work. This is done by measuring the total coverage for an entire test set then finding the smallest subset that has the same coverage, the so-called min-set. Some fuzzers also measure the additional coverage due to mutations to decide whether a particular test case is worth keeping and mutating further.

Target management

Once upon a time fuzzers were required only to generate test cases. Nowadays, expectations are much higher, and some management of the target process is also required.

In particular, it is generally a quick win for the fuzzer to run the target under its own debugger when processing test cases. This way, any crashes that are caused can easily be detected, analysed and triaged. Modern debugging APIs make it straightforward for the fuzzer to save a crash dump and quickly gain knowledge of the important details following a crash, such as the address of the faulting instruction, what kind of operation was being performed at the time, the state of the registers and call stack, etc. This information can be very useful for automatically gauging how likely the bug is to be exploitable, and how best to prioritise it in the set of findings for manual investigation.

As previously mentioned, a fuzzer might also want to keep track of code coverage and execution paths taken within the target process. This may be as simple as tracing (or branch-tracing) the debugged process and logging the addresses of all functions or basic blocks as they are reached. A less computationally expensive approach might be to set breakpoints at the beginning of each function or basic block, removing them as they are reached to gain an overall view of which parts of the code were and weren't exercised (at the cost of losing knowledge of the order in which they were hit). Information such as this will typically be fed back to the fuzzing engine, once the target has crashed, exited or timed out. Armed with this information, the fuzzing engine can decide whether to pursue the avenue of mutation that was used to generate this particular test case, or to move on to other areas.

This analytic data can be made even richer with the use of target instrumentation, in which the target's executable code is modified or augmented to provide additional information about the flow of execution, or the state of memory and registers when certain locations are reached.

Instrumentation may be performed against the target's source code if available (either prior to or during compilation), or against the compiled image as necessary. If the researcher only has a compiled binary to work with, it may be preferable to create a modified version by instrumenting the target image on disk, or by applying the hooks/patches on the fly within the running process, using library injection or a similar technique.

In some cases, instrumentation can even be used in place of debugging. By instrumenting the target to report information regarding its execution, control flow and/or completion to the fuzzing engine in real time (e.g. via shared memory, or a named pipe/domain socket), the overheads associated with using a debugger can be avoided.

Handling crashes

When a researcher successfully causes the target process to crash during fuzzing, the first follow-up action will generally be to ensure that the fault can be reproduced. Often this is as simple as resubmitting the test case for execution, but sometimes the target will behave inconsistently even when operating on the same data. Fuzzing of stateful targets, such as network servers, is generally more complex than that of self-contained programs that operate on files, particularly when the network protocol is made more complicated by with authentication, encapsulation, session handling and such. In these cases, one's hopes of reliably reproducing a crash would be much lower.

Failure to reproduce a test case could occur for many reasons, such as use of uninitialised memory within the target, interaction with external data sources (e.g. the network), availability of system resources or race conditions. It is also common for stateful target applications to respond to a sequence of separate test cases, sometimes making it necessary to consider any residual effects caused by execution of earlier test cases within the fuzzing run. If a crash can't be reproduced, then the researcher may need to resort to analysing the crash-dump to determine the root cause. In practice, such cases are often disregarded as the lack of reproducibility often translates to reduced reliability when producing the final exploit – either because the bug can't reliably be triggered, or because the process becomes unpredictable or unstable when this occurs.

As well as establishing reproducibility, upon detecting a crash a fuzzing engine will generally want to perform some other triaging. Rather than treating all crash-causing test cases as being equally interesting, the fuzzer will want to gain some estimate towards likely exploitability and uniqueness, to save valuable effort during manual follow-up.

Uniqueness can be quite reliably determined by considering the address of the faulting instruction and the call stack of the faulting thread. If two distinct test cases produce crashes with identical stack traces then they are likely to correspond to the same bug, and only one needs to be retained for manual analysis. Conversely, if they crash at a common location but have different call stacks then it is very possible that they are distinct bugs worth investigating separately. This is rather common when the faulting instruction lies within a common routine such as memcpy.

Given a new, unique crash, the fuzzer will finally want to perform some rudimentary exploitability analysis. Inspection of the faulting instruction and the register contents will generally provide enough information to quickly rule out certain classes of bug. For instance, a divide-by-zero exception is generally less interesting than a null dereference, which itself shows far less promise than a write access violation near the stack.

Manual investigation of crashes can be a time consuming process, so the quality of this automatically-generated information can greatly impact the productivity of a fuzzing effort – particularly where the number of crashes is high. Ultimately, the researcher will want the fuzzer to maintain a list of interesting test cases as it runs, complete with sufficient information to quickly get a measure of the type of crash involved and its potential for exploitation.

Some platforms offer development and debugging tools which can be of real benefit when fuzzing, such as Heap Tagging and User Mode Stack Tracing on Windows. These OS-provided settings can be configured per process, and cause useful metadata to be maintained by the heap manager when executing the target process. Using the output of these tools, the otherwise difficult process of determining the owner of a given heap buffer following crash becomes almost trivial. In a similar vein, AddressSanitizer[34] is an LLVM and GCC compiler module that inserts heap manager instrumentation into a program for the purpose of detecting memory errors. Provided target source code is available, it can be used at little cost to pick up on a variety of exploitable faults, including ones that may not have caused a crash during fuzzing (such as off-by-one overflows and use-after-frees).

Test case reduction

The fuzzing process can cause the accumulation of a considerable amount of unnecessary data into generated test cases, and the data responsible for causing the crash isn't always obvious. For example, when fuzzing scripting languages or HTML, a test case may contain thousands of lines, only a few of which are necessary to reproduce the behaviour. Rather than investigate this manually, it is often beneficial to automate the process of reducing the output from the fuzzer to isolate a minimal test case that induces identical behaviour. The principle is simple: sections of the test case's data are iteratively removed and re-evaluated within the target in order to determine their necessity. Often a divide-and-conquer algorithm is used for performance reasons, with the final result being a minimal test case that is much more manageable to analyse by hand.

[34] https://github.com/google/sanitizers/wiki/AddressSanitizer

Hardware fuzzing

While the discussion so far has primarily revolved around fuzzing of software, most of the ideas and techniques are also appropriate for hardware and embedded targets. However, these platforms come with a new set of challenges, particularly during triage and exploitability analysis.

Not least among these challenges is the fact that debugging tools for hardware platforms are typically far less mature than those for software, and in some cases such tools are difficult to obtain or simply non-existent. In these situations, target monitoring management reduces to observing the device for resets or lock-ups. It is usually possible to use automated methods to determine the occurrence of a device reset (for instance, by monitoring the potential of certain pins on the processor or chipset), but knowing where to go from there can be more of an art than a science. Embedded systems are less likely to have source code available, and it isn't always clear which component has faulted. Even if the exact cause of the fault is well known, refining the test case into anything more elegant than a denial-of-service attack can be a complex affair.

Fuzzing enhancements

Depending on the target, the logistics of fuzzing may lend themselves to parallelisation. Particularly when fuzzing a programme that operates in isolation on self-contained data, scaling the fuzzing work across multiple machines may be as simple as dividing the generated test cases up equally among the worker hosts and collating the results as they are fed back. However, some complexities do arise, particularly when the fuzzer itself has a degree of state to maintain (such as in generational fuzzing), although these problems are fairly typical of any distributed computing challenge.

Modern fuzzing has also evolved to take advantage of virtualisation technologies to perform cross-architecture fuzzing using hypervisor-based emulation. This can bring performance benefits where establishment of target state is a bottleneck (for instance, when fuzzing an operating system kernel) as restoration of a memory snapshot can sometimes be quicker than reinitialising the target. Use of a hypervisor can also enhance dynamic instrumentation capabilities by leveraging virtualisation extensions for software traps. Furthermore, it is also possible to take advantage of high-performance consumer hardware when fuzzing code that is engineered to run on low-end CPU platforms (such as embedded systems), through parallel execution using a virtualisation platform such as QEMU[35].

There are also applications of fuzzing to security vulnerability beyond just memory corruption. These applications are still highly fertile and are only now just beginning to be publically explored.

Conclusion

Fuzzing technology, techniques and tooling continues to evolve and remains one of the go-to techniques for vulnerability discovery. While the density of trivially discoverable bugs in mature software is rapidly declining, as well as those trivially exploited, there are still high impact vulnerabilities discovered using well-known fuzzing techniques.

High-profile vulnerabilities are being discovered by more sophisticated fuzzers and while the state of the art has come a long way in a short time, fuzzing as a technique promises to continue to push the boundaries.

[35] http://wiki.qemu.org/

Closing Summary

This paper has looked at the topic of software vulnerability discovery in 2016 as seen by those at the applied end of the security industry. It is our opinion that whilst much progress has been made in the last twenty or so years, there is still much to be done around enhancing consistency, coverage, automation and techniques.

One challenge that is still largely unanswered in a generic, automated and scalable manner is the understanding of systems and their inter-relationships. We see significant fragmentation in architectures, programming languages, frameworks and operating systems as never before. This means that the initial enumeration process of understanding trust boundaries and data flows still remains primarily a job for skilled humans.

As system security improves through the widespread deployment of Trusted Computing Bases that prevent or complicate low-level access to certain systems, the cost of discovery is increasing. Furthermore, as many systems are increasingly reliant on cloud architectures which are often opaque in terms of hardware and software, the ability to replicate these environments in laboratory settings to facilitate analysis is also increasingly challenging, if not at times impossible. This is not to say that these developments will in general preclude discovery of vulnerabilities, thanks to techniques based on virtualisation, static analysis, and other approaches discussed in this paper. However, the level of complexity and investment is generally increasing for more mature software built for more modern operating systems.

Finally, we expect significant gains to be made with the application of machine learning to vulnerability discovery based on the research and initial results[36] [37] [38] seen in the past five years.

Thanks to

The authors would like to thank the below:

Matt Lewis, Richard Turnbull, Michael Tracy, Graham Bucholz, Addison Amiri, Matthew Braun, Christian Prickaerts, Lucas Rosevear and Sherief Hammad.

[36] http://www.vdiscover.org/report.pdf
[37] https://www.usenix.org/legacy/events/woot11/tech/final_files/Yamaguchi.pdf
[38] http://seminaire-dga.gforge.inria.fr/2015/20151009_KonradRieck.pdf

CONTACT US

0161 209 5200
response@nccgroup.trust
@nccgroupplc
www.nccgroup.trust

www.nccgroup.trust
@nccgroupplc