nccgroup
freedom from doubt

RESEARCH INSIGHTS

Exploitation Advancements

www.nccgroup.trust

CONTENTS

AUTHOR

AARON ADAMS

Aaron is a security researcher at NCC Group, where he researches exploit development techniques and builds tools to assist internal consultants.

He has experience with exploit development on numerous platforms. Prior to NCC he worked doing mobile security research at BlackBerry, and threat analysis and reverse engineering for Symantec.

INTRODUCTION

For over forty years the computer industry has been engaged in a cat and mouse game of defensive and offensive techniques and countermeasures. Traditionally, the offensive side almost always has a technological and time advantage.

Exploits are among the primary tools of the offensive side. An exploit is typically a piece of software, or some logic used by an attacker, which takes advantage of a bug or behaviour in the targeted software or hardware. Use of the exploit allows the target to be manipulated in ways unintended by the designer. This manipulation can in turn allow security bypasses, such as executing arbitrary code when only strict program interaction was intended or extracting sensitive data without authentication.

A person writing an exploit only needs to spend as much time as it takes to find the one way in through the various defences in place, whereas a person writing the defence has to spend as much time as it takes to think of every possible logical way around what they're building. This gave exploit writers an edge for a very long time, as writing exploits was often not as complex as one might expect; but it is always becoming more difficult.

This arms race exists in all facets of technology, from hardware to software, from C to Python, but the fiercest competition continues in memory corruption exploitation, typically against software written in C and C++.

Historically, exploit writers and offensive researchers tend to be aware of a significant number of techniques that could be used to overcome future defence technology and mitigations. With many of the waves of defensive mitigations introduced by operating systems, exploit writers immediately knew how to overcome the defences without even doing any new research. The tricks to overcome the defences might have been considered advanced when first discovered, but once the mitigation was in place, the technique would quickly become the norm and well understood by many.

In the last decade and a half, we have seen a significant shift in the defensive realm, with the introduction of many mitigations into mainstream compilers and operating systems, and into their services and applications. This increase in defences has led exploit writers to start leveraging new techniques, along with many that were previously known but considered advanced and unnecessary, in order to achieve a successful compromise.

The defences that attackers will now, depending on the scenario, routinely bypass, circumvent, or purposefully avoid dealing with during exploitation include:

- Address space layout randomisation (ASLR)
- Non-executable memory
- Executable but non-readable memory
- Stack cookies and variable reordering
- Heap metadata hardening
- Heap layout randomisation
- Delayed heap freeing
- Object allocation partitioning
- Exception handling: SafeSEH, SEHOP
- Pointer encoding
- Sandboxing
- Input filtering
- Supervisor Mode Execution Prevention
- Vtable integrity
- Control flow guard

Not all of these mitigations have an easy workaround for an attacker, so often the situations in which they would pose a problem are simply avoided. Take stack cookies and variable re-ordering as a prime example. Without an information leak (a way to retrieve information from the target process either locally or over a network before exploitation), these mitigations can make many stack overflows difficult, if not impossible, to exploit. However, following the path of least resistance means that attackers spend their time trying to exploit other bug classes like type confusion, heap-based buffer overflows, or use after frees, as an alternative.

At one point it was considered advanced to understand these defensive technologies and be able to defeat, circumvent, or avoid them while still leveraging a bug. As more and more exploit techniques are used routinely, the techniques start to lose their advanced status and new, more esoteric, tricks take their place, as we see with many of the zero-day exploits discovered recently.

Modern Exploitation

What many people think of as advanced exploitation techniques have in fact often been known and even practically deployed for over a decade. The differentiating factor is that the techniques weren't necessary for everyday exploitation scenarios. A variety of techniques have moved from the advanced category to the normal category, to the point where we see them in almost every major attack.

Information leaks

It is largely accepted by exploit writers that information leaks have become the most important part of a successful attack; some have even dubbed it the information leak era. In the past, because of failure to implement ASLR effectively, there was often no need for information leak to achieve successful exploitation. Now that ASLRd processes, especially on 64-bit operating systems, have reasonably random memory layouts, exploit writers must resort to leaking addresses from the process to inform the rest of the exploit. Almost every major memory-based exploit now relies on some form of information leak.

Almost every major memory-based exploit now relies on some form of information leak.

ASLR is not the only reason one might need an information leak. You might need one for finding executable modules in memory to facilitate return oriented programming (ROP), determining the layout of objects in memory to know where to corrupt, reading secret cookie values to bypass mitigations; finding key data structures to continue a process cleanly after exploitation has completed, and many more reasons.

These leaks are typically used in two ways:

1. An exploit writer leverages the original vulnerability they want to exploit in order to build what is called a leak primitive. This will often result in triggering the vulnerability many times in order to leak various areas in memory.
2. A separate vulnerability is used for the information leak. Sometimes a bug that lets you eventually gain code execution isn't sufficient for leaking information, so you are forced to use a separate bug.

Sandbox escapes

Sandbox escapes are among the areas that have seen the most advancement in recent years. The general idea of this is quite old, as exploit writers have been breaking out of primitive chroot jails for a long time. But as sandbox technology has advanced and arrived on the desktop, so too have the techniques required to break out of them on more modern operating systems.

In client-side exploitation scenarios, sandbox breakouts have become a necessity for exploit writers, as almost all mainstream browsers and document-parsing tools use some form of sandbox. One point worth mentioning, however, is that breaking out of the sandboxes themselves doesn't always involve some new or advanced exploit techniques; it simply involves an additional exploit. The requirement of chaining multiple, often completely unrelated, exploits in a single attack at a more abstract level represented major sophistication in the past, but again has now become fairly commonplace.

The purpose of a sandbox is to limit the environment in which an attacker finds themselves after the first stage of successful exploitation. Were a browser compromised and an exploit to obtain arbitrary code execution carried out, the attacker might be executing in an environment with no meaningful network, filesystem, or system access. This forces them to resort to breaking out of the sandbox. The most common approach to this is to use a second exploit that targets the operating system kernel.

Depending on the design, other breakout exploits, which target the sandbox broker processes or, if present, the hypervisor, are also seen. Many interesting and sophisticated techniques have come out of these breakouts; some of the most notable being from exploit competitions, game console hacking and jailbreaks, rather than malicious exploitation. We do still see new and interesting attacks in this space, such as leveraging an identical bug in different privilege contexts to achieve both client-side exploitation and sandbox breakout.

Malleable bugs

Much of modern exploitation involves attacking very specific bug classes because they exhibit properties that facilitate bypassing many mitigations. Although exploit writers create innovative ways to leverage restrictive bugs to do what they need, there is an increasing requirement for a bug to exhibit certain behaviors to facilitate bypassing all of the modern mitigations. If it doesn't, it will either be ignored in favor of a more malleable bug, or be put to use as one of a collection of bugs used to leverage an attack.

At one point, before ASLR became so effective, a bug that allowed an arbitrary write to any location in memory was often seen as favorable. There was almost a simple recipe one could employ to exploit it. However, in the modern age in which no static addresses are known in advance, this type of bug isn't always ideal. Instead, small linear overwrites, with minimal data restrictions, have become much more favorable than arbitrary writes. This is not to say that the eventual goal isn't to construct an arbitrary write; however, in modern exploitation scenarios a smaller controlled linear overwrite in combination with heap feng shui (massaging) can give you a much more favorable starting position that lets you slowly build up a collection of exploit primitives. It's worth noting that heap feng shui is also an exploit technique that used to be considered quite advanced, but has just become part of the modern toolset.

Exploit releases from malware

In the past there was a fairly vibrant community of exploit writers who would release their work to the public. We've now seen this habit change, and malware will now leverage zero-day exploits, exploits for bugs that had yet to be proven exploitable publicly, or for known exploitable bugs for which no public exploit had been available. The reason for this shift is at least in part due to the exploit community largely withdrawing from the public eye, leading to malware developers needing to develop their own private exploits. Another likely cause is the ongoing monetisation of malware and exploit technology, which allows malware authors to purchase exploits to plug into their software as needed.

In the past there was a fairly vibrant community of exploit writers who would release their work to the public. We've now seen this habit change.

This is possibly evidence that the increased difficulty of exploitation, which leads to a larger time investment, prevents some hobbyists from being able to exploit the bugs in a reasonable amount of time and that they might be less willing to give away the work for free. On the same note it shows that the level of sophistication of some malware authors is increasing, in that many of them no longer rely on adapting publicly-available exploits. Public research is often enough for them to build upon to develop their own exploits, and in some cases the malware authors are now leveraging exploit techniques that had not been shown publicly at all.

Advanced Exploitation

The term "advanced" is subjective and is a window looking out over a moving landscape. The exploit techniques already described were once advanced and are now fairly standard. It is interesting to consider what could currently be considered advanced exploit techniques.

The exploit techniques already described were once advanced, and are now fairly standard.

One aspect of modern exploitation is that more aggressive defences, and a better general understanding of low-level technologies, seem to have pushed research and techniques to what is almost the fringe of what can be classified as an explicit hole or flaw. What is especially interesting about these techniques is that not only do they start to blur the line of traditional vulnerabilities, but in many cases the vendor response is to not fix the underlying issue; sometimes software vendors are unable to even fix the bug, and other steps must be taken to mitigate. Similarly, some bug classes transcend the thinking of a typical software vulnerability and exploit a more abstract design as a side channel.

Although many of the following bugs and exploit techniques represent areas that are hard to fix, whatever is pushing researchers and attackers to find such vulnerability classes could be indicative of vendors starting to succeed in some of their defences. As a general rule, as bugs get harder to find or exploit, people are pushed to more obscure and extreme ways of achieving their end goals. This often results in fascinating research and new areas for security hardening, but also tangible risks to those trying to secure their infrastructure.

DRAM row hammering

An interesting physical property of dynamic random access memory (DRAM) is that the aggressive use of certain rows of memory cells can result in abnormally fast discharging of capacitors in the rows of cells adjacent to those being hammered, which, given the right timing, can corrupt those cells by triggering what is known as a "disturbance error": causing bits to flip from one value to another. This is known simply as row hammering. In 2015 it was shown that these row hammering disturbances could be abused reliably by native code, which leveraged specific cache flushing instructions on some hardware to break out of the Google Chrome sandbox and to manipulate Linux kernel data in order to elevate local privileges. Newer research has suggested that row hammering can be reliably triggered from JavaScript without even requiring the direct execution of a cache flushing instruction.

Row hammering exploitation leverages physical properties of RAM. A software vendor can't fix this vulnerability, but they can reduce the availability of certain functionality that can help exploitation. The Chrome browser sandbox no longer allows execution of the cache flushing instruction on x86. The Linux kernel now prevents an unprivileged user from being able to query the underlying physical frame number for a given allocation, as this information was used to inform exploitation of row hammering. These mitigations will help slow down attackers, but in the end don't fix the underlying issue. The only solution for users is to buy higher-end hardware that either has built-in mitigations, such as more aggressive row refreshing timing, or error correcting codes to detect unwanted bit flips. Any computer that is not, or cannot be, physically upgraded is permanently vulnerable. The practical abuse of this type of vulnerability is a great example of modern advanced exploitation.

Use MemoryProtect to defeat ASLR

In 2014, Microsoft's Internet Explorer browser deployed a new mitigation called MemoryProtector, which is designed to hamper the exploitation of use-after-free (UAF) vulnerabilities. UAF bugs have become one of the most popular client-side vulnerabilities to exploit in recent years.

In 2015 it was shown by researchers at the Zero Day Initiative and Google Project Zero that this MemoryProtector mitigation could, in two different ways, be used as an information oracle to bypass the ASLR mitigation. This scenario presents an interesting problem for a vendor, because the original mitigation does in fact serve a purpose -- to hamper the exploitation of certain bug classes -- and therefore is still a valuable piece of technology. This is another interesting case of exploit writers pushing advancements towards the fringes of what constitutes a weakness or vulnerability, where a vendor must weigh the value of preventing certain bugs versus enabling the easier exploitation of other bugs. In this case, the vendor has so far decided that the benefit from MemoryProtector was more important than the weakness it presents to the ASLR mitigation.

KASLR timing attacks

Kernel ASLR is a mitigation deployed by a few operating systems to hamper the ability to exploit vulnerabilities that need to know where something is located in kernel memory. This might be, for example, the location of a function pointer to overwrite or the location of a kernel payload an exploit needs to execute. This has in turn increased the number of information leak vulnerabilities being found and fixed in kernels. Not only is this information being removed in the form of bug fixes, but also sandboxing is being used to reduce the ability of a compromised process to reveal information that may be otherwise accessible, even if not as a direct result of a vulnerability.

Despite all of these efforts, we see another exploitation technique on the boundary between bug and expected behavior. By understanding how memory caching works on a processor, specifically page faults and the resultant translation lookaside buffer (TLB) caches, it is possible to use subtle timing differences exhibited by the CPU as an information side channel to discern between addresses that you can't even directly access.

Although timing attacks are somewhat probabilistic leaks compared to an explicit one that might be triggered from a more traditional software vulnerability, the technique can be perfectly effective and exists on the fringe of software and hardware. It is not functionality that can be mitigated through software changes, unlike a more traditional vulnerability. Currently exploit writers aren't leveraging these timing attacks, as it's typically far easier to find an information leak bug, but once the bug well dries up, this more advanced technique might become the norm.

It is not functionality that can be mitigated through software changes, unlike a more traditional vulnerability.

Self-mapping page table entries

In 2014 increased attention was given to a feature of page table handling on some operating systems, called self-mapping, in which a given index within a page table will actually reference back to the physical address of the page table itself. What this means is that, given a userland virtual address, you can make some static modifications to the virtual address to create a new virtual address, which will only work in kernel mode, but which will resolve to the physical address of the page table entry that manages the physical page backing the original virtual address.

Advanced Exploitation
Cont...

Why is this useful? Assume you have an arbitrary write in kernel space and want to be able to execute an exploit payload in userland. Also assume that the supervisor mode exploit protection (SMEP) mitigation prevents you from just jumping directly into executable memory in userland space. Modern kernel hardening also means that the locations in kernel memory in which you can store your payload are non-executable. One option is to use your arbitrary write to manipulate the page table entry for an address holding your payload directly, either in userland or in kernel space. If the payload is in userland an exploit could modify the associated page table entries to mark the address a supervisor range, meaning executing data stored at this address from kernel space will no longer trigger the SMEP mitigation. Similarly, an exploit could modify the page table entry of a read-write memory location in kernel space, and mark it as executable. This way execution could be redirected, without violating SMEP, and the payload can actually be executed.

Although in theory this problem could be mitigated in some ways, it is also a legitimate and intended feature of many page table management designs, and has been for decades, as it allows a kernel to make changes to page table entries rapidly, without deploying more expensive table lookups each time.

Virtualisation security introducing insecurities

An increasingly popular form of sandboxing is to leverage virtualisation technology to keep parts of a system more heavily isolated. A good example of this is the Qubes OS, which leverages the Xen hypervisor to run programs within their own isolated operating system environment.

One interesting aspect of some virtualisation technologies such as Xen is that they fundamentally change the environment in which an operating system would normally run, in order to facilitate certain virtualisation goals. What can happen in translation is that certain key security technologies available on a non-virtualised environment, such as the SMEP and supervisor mode access prevention (SMAP) mitigations on Intel processors, become unavailable within the virtualised environment. Specifically, in the case of a paravirtualised Xen guest machine, the entire guest operating system is run in ring 3, which means that the guest kernel cannot enforce SMEP or SMAP, as it explicitly requires running in ring 0 to be effective.

An increasingly popular form of sandboxing is to leverage virtualisation technology to keep parts of a system more heavily isolated.

If virtualisation is being used as a hardening measure for a more complete operating environment, then this might not be a big problem. However in a cloud environment for instance, where a customer might simply have no option but to operate within a virtualised environment, their security is impacted by the convenience of using the cloud technology. A vulnerability that might otherwise be unexploitable thanks to SMEP and SMAP mitigations could still be a perfectly legitimate candidate for exploitation on virtualised environments, and the users of the technology might not even realise that they are at increased risk. Although this has yet to become commonplace, as mitigations become increasingly difficult to exploit, attackers that have a more advanced knowledge of system and virtualisation internals may begin to seek out this type of opportunity.

Abstract interpreter abuse

An interesting case of an ASLR bypass that is more advanced than the typical information leak, is one that involves leveraging how an interpreter might store information within data structures.

This idea has been practically demonstrated by researchers, though it is not yet something people seem to be actively finding or using in their exploits. The premise is that certain types of data structure, such as a dictionary, might be sorted using values taken from the underlying object, and that different object types such as integer values and pointers to values might be intermixed. This intermixing of data can actually be used as a side channel to infer an address, by simply interacting with and inferring properties of some target object within the data structure, based on data you're actively inserting into the same data structure.

...many attackers will favor leveraging their corruption bug to build an information leak, or stick to path of least resistance

Although this type of ASLR information bypass hasn't become common place yet, as many attackers will favour leveraging their corruption bug to build an information leak, or stick to path of least resistance methods such as heap spray, I think eventually this type of flaw will be used more often. This type of information leak is much more difficult to find using automated analysis and compiler checks, as it is a more abstract problem that is not caused by traditional bad coding practices.

Out of order execution engine side channels

Although CPU-cache-based side channels for data exfiltration have existed for some time, there is new research being done in this realm as well. It was recently shown that a processor's out-of-order execution engine can be used to exchange data between two co-resident virtual machines.

The out-of-order execution engine is used by a CPU when it is processing opcodes, the single machine instructions to which a native application is compiled down. Typically, a CPU would fetch each new instruction to be executed and place it into an ordered pipeline. However, there are inefficiencies with this, as certain instructions can cause stalls that prevent the next instruction from being executed immediately. To counter this, many processors will re-order certain instructions in order to maximise the efficiency of the pipeline.

In 2015 it was shown that this re-ordering behavior could be abused by two collaborating systems on the same hardware, but in different virtual machines with different security policies, to exchange data that would violate the policies on one system. Although this is specifically related to data exfiltration, rather than traditional exploitation, it is another good example of advanced attacker-oriented research moving towards more obscure, low level, and fringe areas of research that become increasingly difficult to address from a defensive standpoint.

Future Exploit Techniques

We see vendors and researchers increasingly making an effort to combat popular exploitation techniques and bug classes. In many cases this new research is simply improving on theoretical mitigations and security first demonstrated decades ago, but which were not adopted or suffered from performance issues that are only now being addressed due to the increased necessity for a solution.

We see vendors and researchers increasingly making an effort to combat popular exploitation techniques and bug classes.

For example, control flow integrity (CFI) will significantly impact return-oriented programming as a common exploitation technique. CFI is now being introduced via Control Flow Guard in recent versions of Microsoft Windows, and the LLVM compiler has also recently added support for CFI. However, we also see that exploit writers have already started to find the path of least resistance, as it has been shown that just-in-time (JIT) compilation engines don't work well with effective control flow analysis, and thus an attacker can bypass control flow mitigations altogether by targeting JIT.

Similarly, some use-after-free and type-confusion attacks are starting to be targeted by the introduction of vtable cookies. This prevents one object with a specific vtable from being operated on when the underlying memory has changed, because the associated code can tell that the vtable is incorrect. This will likely lead to increased discovery techniques and bugs that leverage objects that don't have such protections, or simply targeting software that has none of these protections. Eventually it might cause yet another class of vulnerabilities, the next easiest to exploit, to surge in popularity.

In general we'll continue to see exploit writers taking the path of least resistance as each new mitigation is introduced. If software X becomes hardened, software Y will become the new target. Attackers have continued to target Adobe Flash in the last few years because it's one of the easiest targets to exploit; before that Java was a principal target. Flash has recently been hardened against one of the features that made it so ideal, so perhaps exploit writers will move on to something new.

As more techniques and bug classes are mitigated completely, we will start to see a move towards even more of the techniques on the fringe of software and hardware design. These will not only be increasingly difficult to fix, but in some cases might not be fixable at all, because the way in which the flaws can be abused is also an intrinsic part of what makes the design useful for non-malicious purposes in the first place.

As more techniques and bug classes are mitigated completely, we will start to see a move towards even more of the techniques on the fringe of software and hardware design.

Conclusions

The exploits leveraged by attackers are becoming increasingly sophisticated, but typically only to the minimum level required to get the job done. The theoretical and esoteric attacks of a previous era have become the requirements of modern exploitation, and in many ways they should no longer be considered advanced. In their place are a new set of theoretical and esoteric attacks waiting in the wings until the time is necessary for exploit writers to leverage them more aggressively.

As has always been the case since dawn the of the offense vs defence dance, the primary key to hampering attackers is to increase the investment they must make in order to pull off a successful attack. This means increasing upfront investment in developing their exploits, by proactively finding bugs and testing your own software, introducing mitigations to hamper exploitation of issues you don't find, and adding layers of security to slow down the exploitation of bugs that will inevitably be missed. This extends beyond just the software on external facing systems, into all software and hardware of the entire infrastructure deployed by a user or company. Every layer that an attacker encounters must be a new hurdle to slow them down.

Systems will become harder to exploit, but a determined attacker, through the exploitation of human error, software bugs, or logical errors, will always find a way to exploit the security of a system. It is up to everyone: vendors, software developers, users, and companies, to ensure that they design, configure, and deploy their technology in ways that make the attacker's job as hard as possible. This should be done in an effort to make the time investments become discouraging enough to not be worthwhile, or to make successful attacks result in less exposure of information, assets, and control than the attacker had hoped to obtain.

CONTACT US

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

United Kingdom	Europe	North America	Asia Pacific
Manchester - Head office	**Amsterdam**	**Atlanta**	**Sydney**
Basingstoke	**Copenhagen**	**Austin**	
Cambridge	**Luxembourg**	**Chicago**	
Cheltenham	**Munich**	**New York**	
Edinburgh	**Zurich**	**San Francisco**	
Glasgow		**Seattle**	
Leatherhead		**Sunnyvale**	
Leeds			
London			
Milton Keynes			
Wetherby			

www.nccgroup.trust
@nccgroupplc