Systems Security Reflective Report

Nashe Mncube (27604) David Sharp (36688) Daniel Davies (35349)
Chetankumar Mistry (38523)

1 Introduction

In this report we will perform a reflective analysis of the Buffer Overflow Attack. This will involve discussing the countermeasures that currently exist against the attack in modern systems. We will try and provide an overview of the research space that we believe to be most relevant to the attack, and reflect on the consequences of this research.

2 Countermeasures

In this section we discuss several countermeasures that exist against buffer overflow attacks.

2.1 Address Space Layout Randomisation

Address Space Layout Randomisation (ASLR) aims to increase the security of the system by providing a random memory space every time a program is executed. ASLR was created in 2001 by the PaX project as part of Linux and first added into Windows with Vista in 2007 [1]. To explain ASLR we will describe its implementation in Vista [2]. An image compiled with ASLR is given a random offset from 256 options, the image is then loaded at this offset. The stack and process heaps are then each loaded at one of 32 possible locations, then the initial stack pointer is decremented by a random amount of up to half a page.

All of these randomisations put together means that it becomes extremely unlikely for an attacker to know the layout of the program's memory during execution and therefore unlikely for the attacker to be able to induce code execution using the return address.

ASLR aims to reduce the likelihood of an attack succeeding, but if an attacker could predict the memory layout the attack would still work every time. In order to more robustly prevent attacks the return address needs to be defended on the stack in some way.

2.1.1 Brute Force Attacks on ASLR

ASLR relies on the memory layout being sufficiently random that an attacker cannot predict the layout. On 32-bit systems the memory cannot be possibly mapped to enough areas to be robust against brute force attacks [3]. When the only defence on a 64-bit system is ASLR there are still enough possible memory locations to delay brute force attacks to several hours rather than the milliseconds required on 32-bit systems. This relates to the computational complexity of performing an attack on an address space. For a 32-bit address space we have approximately 4e+9 possible addresses. For a 3GHz processor, which is roughly standard, assuming each operation takes one-unit of time we can perform a brute force attack with reasonable time guarantees. However for a 64-bit address we know have approximately 1e+19 possible address spaces.

2.1.2 *offset2lib* Attack

Another attack on ASLR is *offset2lib* attack. *offset2lib* was a vulnerability discovered in the Linux implementation of ASLR [3], specifically *offset2lib* is the invariant distance between the load location of

the application and the load location of the libraries. The value depends on the set of libraries used by the application and the version of each library, but it's calculable offline. Knowing the value of *offset2lib* allows an attacker to work out the memory mapping and launch a shell.

This exploits a weakness in ASLR when programs are PIE(position independent executable) compiled. PIE compiled simply means that the compiled program can execute properly regardless of where it is placed in memory, thereby allowing ASLR. ASLR then works by loading the executable as if it's a shared library. Any other shared objects are then loaded one after the other. This means that if a shared object has a memory leak, one can de-randomize the entire address space. These shared objects may be shared libraries.

2.2 StackGuard

StackGuard is a two component method that aims to detect when buffer overflows occur modifying memory out of bounds, and if so switch to a higher security level that makes the return address read-only [4]. These two components are stack canaries and memguard.

2.2.1 Canaries

A relatively simple way of protecting the return address from buffer overflow attacks is to place a known binary word next to the return address on the stack, once the function is ready to return just check whether the known word is still the known word, if it is then this implies that the return address hasn't been affected by a buffer overflow and is safe. This is a presumption as an attacker who may know the value of this word can ensure that it is kept the same when writing over it. If the the known word has been altered then the return address cannot be presumed safe and so the best course of action is normally to kill the program. This is value is called a stack canary.

Due to the nature of buffer overflow attacks it isn't an option to skip over the value of the canary when changing the value of the return address which makes this defense somewhat successful. However an approach where the attacker discovers or calculates the value of the canary and in so doing can tailor the buffer overflow to leave the canary on the same value, thereby avoiding detection, is a possibility. To protect against this a randomised canary can be used so the attacker must be able to inspect the memory image of the program *during execution* in order to simulate the canary in their overflow. This assumes that the generation algorithm of the canary value is random enough so that an attacker struggles to compute the value.

By using a canary, StackGuard can detect whether or not the return address has been modified on function return but prevention of this modification is obviously more ideal than detection alone.

Canaries also have the drawback of not actually preventing any overflows by their operation, they simply are there for detection depending on their implementation. This means that an attacker essentially has control of the stack frame.

2.2.2 MemGuard

MemGuard is the component of StackGuard that aims to wholly prevent editing of the return address on the stack.

MemGuard works in general by marking pages in virtual memory that contain the protected values (the return address in this case) as read-only; a trap handler then catches any writes to the protected values and emulates those writes to non-protected values on protected pages. This general approach has an extreme performance overhead but is worth it in quiet areas of the address space where writes don't need to be emulated often.

The top of the stack however is not a quiet area of address space so write emulation has to happen very commonly resulting in a significant performance hit. StackGuard makes some alterations to MemGuard to change it from being optimised for protecting kernel space to protecting stack space. StackGuard uses the processor debug registers to cache the most recently protected memory addresses in order to try

and specifically protect only the top most page of the stack, reducing the number of emulated writes that need to occur and reducing the performance impact.

2.2.3 Canary or MemGuard

While a canary approach is far more performance friendly, it lacks the robustness of the MemGuard adaptation. StackGuard therefore dynamically switches between the approaches in order to stay in canary mode most of the time, only switching to MemGuard when an attack is detected to mitigate the performance loss of emulated writes when in MemGuard mode.

2.3 Address Sanitation

2.3.1 AddressSanitizer

AddressSanitizer[5] is a Google made tool which is supported by the Clang, GCC and Xcode compilers which will detect memory-corruption bugs such as buffer-overflows. This feature can be enabled by passing in the `-fsanitize` flag on GCC[6] for example. When a program compiled with AddressSanitizer is executed, any memory-corruption bugs that are detected during runtime will spawn an error message explaining what the error is and why it has happened.

2.4 KernelAddressSanitizer

KernelAddressSanitizer is a feature in the Linux Kernel which will trap and kill any executing process which attempts to dereference a freed pointer (use-after-return bug), or when trying to access an element out of the bounds of a declared array. It works by, during compile time, inserting function calls[7] before each memory access, and these functions check whether or not the memory access is valid or not.

2.4.1 Limitations

Address Sanitation tools cannot yet detect all use-after-return bugs, and also cannot any uninitialised memory reads. Also, adjacent buffers in structs and classes are still vulnerable to overflow, this is to preserve backwards compatibility. AddressSanitizer is also restricted due to the fact that it needs to be manually turned on by using compiler flags.

KernelAddressSanitizer needs to be enabled in the Linux kernel whilst it is being configured. This means that, even if a program tries to be compiled with address sanitation support, if the Kernel hasn't been correctly configured before hand, then the compiled programs will not this feature enabled.

2.5 Data Execution Prevention and Non Executable Stack

The final idea to consider is the most general defence. The key vulnerability that allows buffer overflow attacks to have as much power as they do is that they can manage to cause a program to return from a function call to arbitrary code. If we can change our memory to have the assumption that data should not be able to be executed unless explicitly marked as such then that drastically reduces the power of the attack.

Data Execution Prevention (DEP) consists of hardware and software implementations that checks memory to stop programs executing code from areas they shouldn't[8].

Hardware-enforced DEP utilises processor hardware to mark memory to indicate that code should not be executed from that location, such that processors that support hardware-enforced DEP will raise an exception if code is executed from a non-executable location. However this does require processor support, specifically it requires that the processor supports Physical Address Extension (PAE), which uses 36-bit addresses to allow addressing up to 64GB of memory rather than the 4GB addressed by a 32-bit address. PAE increases the size of the Page Table Entry (PTE) which contains the base physical address of a page and the attributes of that page, in PAE mode the PTE are extended from 32-bit to 64-bit allowing extra space to mark that page as non-executable.

Software-enforced DEP is a lot weaker than the hardware-enforced variant. When an exception is

thrown, software-enforced DEP checks if that exception is registered in a function table for the application that the application is built with. It helps prevent exploits that take advantage of Structured Exception Handler overwrites, and is only really used on machines that cannot support hardware-enforced DEP i.e old 32-bit processor machines. As a compile time change it only supports protecting system libraries and limited third-party programs.

2.5.1 Limitations

Unfortunately DEP alone is not sufficient to fully stop buffer overflow exploits [9]. If an attacker is able to modify the execution flags of the process being exploited then the attacker is able to execute code from non-executable regions. This is possible by using a return to libc attack which doesn't require an attacker to use any shellcode to obtain control, but simply uses functions within the C library. Another way to execute code from non-executable regions is by jumping into ntdll which is a library that has some routines that handle hardware-enforced DEP compatibility, and exports the Windows native API. This attack is only feasible without ASLR active as the precise locations of the libraries isn't known under ASLR.

2.6 Production systems example

2.6.1 WhatsApp Arbitrary Code Execution

A highlight for buffer overflow exploits in 2019 came from WhatsApp. This was particularly high profile as it would allow remote code execution [10] with just a phone call to a target device (without the constraint of the user even needing to pick up [11]).

Security analysts were able to sift through commit differences in patched code to recreate what they believed to be the bug and show Facebook's fixes. The attack allegedly played out as follows [12]:

- The hacker would call the phone of the target
- With carefully selected packets, there were two potential buffer overflow targets that were patched in the latest WhatsApp release. Analysis of the patched code showed the following bugs and corresponding solutions [13]:

- In one area of the code, dealing with packet bursts, a function is vulnerable to integer underflow. If the right packet size is specified by an attacker (greater than `0x7A120` in this case), the integer underflow leads to a packet length check being skipped [12], which ultimately leads to two unchecked `memcpy` functions taking caller defined input. This is the first potential entry point. Analysis of the patched code shows that, for whatever reason, this whole function had simply been removed. [12] (See the vulnerable function here).

- In a function to listen for incoming traffic, length checks on packet sizes had been completely left out. Analysis on the patched version shows a check on this `memcpy` had been implemented, constraining packet sizes to less than `0x5c8` [12]. (See the vulnerable function here).

This example fundamentally shows that, while there are many safeguards in place against buffer overflow attacks, prevention starts with the programmer, and catching these kinds of errors, even in production systems, is a very difficult task. (**NB:** The main aim of this discovery was the installation of surveillance malware to snoop on private data trafficked through the WhatsApp platform [11]).

3 Conclusion

Part of the reason that Buffer Overflow attacks are still a prevalent issue today is that many legacy programs have never been updated to use *safe* versions of common functions like **strcpy** in C. Additionally the attack is fairly easy to set up compared to the benefits of a successful attack i.e arbitrary code execution, which encourages looking for these buffer exploits.

Countermeasures like ASLR, Stack Canaries and Data Execution Prevention together can mitigate many potential buffer overflow exploits but taken alone cannot be assumed to be fully secure.

While the industry has come up with means to limit buffer overflows the systems are not foolproof and buffer overflows will continue to be a problem into the future.

References

[1] B. Spengler, "Pax: The guaranteed end of arbitrary code execution." "https://grsecurity.net/PaX-presentation.pdf".

[2] O. Whitehouse., "An analysis of address space layout randomization on windows vista," 2007.

[3] I. R. Hector Marco-Gisbert, "On the effectiveness of full-aslr on 64-bit linux," 2014.

[4] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, SteveBeattie, Aaron Grier, Perry Wagle, Qian Zhang, Heather Hinton, "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," 1997.

[5] "Addresssanitizer." https://en.wikipedia.org/wiki/AddressSanitizer.

[6] "Using the gnu compiler." https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html.

[7] "The kernel address sanitizer (kasan)." https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html#usage.

[8] Hewlett-Packard, *Data Execution Prevention.*

[9] S. Skape, "Bypassing windows hardware-enforced data execution prevention," 2005.

[10] "CVE-2019-3568." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-3568/, 2019.

[11] "Whatsapp Buffer Overflow Possibilities." https://www.wired.com/story/whatsapp-hack-phone-call-voip-buffer-overflow/, 2019.

[12] JT Keating, "WhatsApp Buffer Overflow Vulnerability Reportedly Exploited In The Wild." https://blog.zimperium.com/whatsapp-buffer-overflow-vulnerability-under-the-scope/, 2019.

[13] Ýmir Vigfusson. https://blog.adversary.io/whatsapp-hack/, 2019.