Software vulnerabilities

A look into the field on memory corruption

Niklas Zabel

Seminar on IT-security
supervised by Prof. Dr. Gerd Beuster
FH-Wedel
Summer Semester 2019

Tornesch 20.05.19

Abstract:

The exploitation of software vulnerabilities is a common occurrence. The field is constantly moving forward with new exploitation techniques being developed to surpass the present security mechanisms. All the while the security field tries to keep up in this constant arms race.
I will go over the most common flaws and mention some problems regarding software development. As well as follow it up with a deeper look into the field of software vulnerabilities using memory corruption as an example. This includes explaining multiple advances in the defense against memory corruption attacks by going over the functionality and weaknesses of non-executable data segments, canaries, address space layout randomization and control-flow integrity.

Contents

1 Introduction

1.1 Introduction

Software vulnerabilities have been a crucial concern for over three decades. Increasing reliance on software in our daily lives only increases our potential attack area. A lot of electronics are connected to the Internet nowadays making them potentially assailable for everyone. Unintended weaknesses in software are very common. The exploitation of software vulnerabilities offers a wide range of possibly malicious things that can be done. Plenty of vulnerabilities are simply caused by insufficient care concerning IT-security. I will consider software vulnerabilities under the following definition: “In computer security a vulnerability is a weakness or flaw in one or more software components that can be exploited to compromise the integrity, confidentiality, or availability of a system and its information resources” (SysSec 2013)[11].

1.2 Goal

I will try to show the difficulty of the field of software vulnerability starting with a brief overview by going into some common vulnerabilities and general problems. Followed by diving into memory corruption by showing some mechanisms that were developed along the way to combat vulnerabilities as well as their weaknesses.

2 Common Vulnerabilities

2.1 Introduction

Software vulnerabilities are flaws in software design or implementation that cause unwanted side effects or allow exploitation of the software. There are multiple organizations that strive to inform about common vulnerabilities like the Common Weaknesses Enumeration (CWE) and OWASP. Both publish information concerning common and dangerous vulnerabilities[4][10]. The following classifications are some broad categories which contain many common weaknesses[11].

2.2 Memory Errors

Memory management can be particularly problematic if not done correctly, with memory leaks not being the only problem. The pursuit of fixing memory corruption attacks has been going on for over three decades[12]. Buffer overflows are a very common and known problem concerning security. Other dangerous flaws include dynamic memory errors like dangling pointers, double or invalid frees, null pointer dereferences.[11]

2.3 Input Validation Errors

Input validation errors happen whenever a function of a program accepts data that it is not made for causing massive problems via side effects. One such example are SQL injections which are infamous for being a huge risk for the security of databases behind web pages. They allow users to execute their own SQL code on the database potentially clearing the whole database. Assuming that users provide valid data is a very dangerous approach to software development. User input should always be checked for validity. Common input vulnerabilities besides SQL injections are code or command injection, uncontrolled format strings, cross-site scripting (XSS) and directory traversal[11].

2.4 Race Conditions

Race conditions describe problems that happen when multiple sources alter and access a resource or data at critical moments causing inconsistent program states. It is generally problematic for programs that use different tasks/parallelism. A common cause is when parts of the program rely on the use of non local information that can be altered by other sources.

If, for example, we have a variable like *stockOfLemons* and we want to use five lemons to make lemonade, we first need to check whether we still have five left. In this case, the problem is caused when both tasks do this at the same time. If both of them check whether there are still five left before any one of them takes the five they need away, we run into the problem that both think they can take five. At this point the tasks attempt to subtract 10 from *stockOfLemons* even though there might only be six. We now have a *stockOfLemons* with a value of -4 which is an invalid state (this is just a minor example to explain the concept). Interrupts further increase the likelihood of these events. Common problems resulting from race conditions are simultaneous access and time-of-check-to-time-of-use (TOCTOU) bugs[11].

2.5 Privilege-Confusion

The common web vulnerabilities of this type are Cross-Site Request Forgery (CSRF) and clickjacking[11]. CSRF abuses web forms by using a static form from another website sending requests with the same format and abusing the fact that the user may be logged into the other site. The requests are sent by tricking the user into pressing a button. Clickjacking embeds another site into itself and causes the user to do unwanted actions on said site like clicking a button.

3 General Problems

Software vulnerabilities are still very common, even though most of them are theoretically avoidable. The development of bigger commercial software projects introduces plenty of problems concerning security, with miscommunication being a major one. Additionally, time to market is vital, where as security plays more of a background role in most cases with functionality being the main focus[11].

The deployment of security patches is also not a trivial task. Companies value stability of their IT-infrastructure and generally rely on a lot of different software which may all be interconnected. Updates of any kind can cause massive problems concerning other programs. Patches sometimes cause bigger problems than the ones they attempted to fix.

One component that is reduced by the increasing speed of modern computers is performance. Most quick solutions to vulnerabilities introduce new load on the system which, in turn, causes a reduction in availability of the system in favor of integrity. These side effects may be detrimental. Long loading times decrease user engagement and some of it can be leveraged by delegating tasks to be done during compile time but these also have to be considered for software development as it is unproductive time.

The complexity of attack strategies only keeps increasing as new barriers are implemented to block attackers. New techniques to combat memory corruption generally only stop certain exploitation techniques but not all. Attackers will generally find new and more complex strategies to achieve their goal.

4 Memory Corruption

The exploitation of memory vulnerabilities is very common. “As of 2016, about 86% of all vulnerabilities on Android are memory safety related.” (“Control Flow Integrity”, Android, 2018, para. 1)[2]. Memory corruption can be used in a variety of attacks to achieve different goals. The most frequent attacks are those that aim to execute specific code using the permissions of the application and are commonly used by malware. Data-only attacks aren't covered under many defensive techniques, because they only aim to modify critical data of a program without altering the control-flow.

Despite the fact that the field of IT-security concerning memory corruption has been developing for over three decades, proper solutions to completely eliminate memory corruption in a satisfying way are still to be found. Most defenses are developed to stop existing exploits.

4.1 Morris Worm

The 2nd of November 1988 was an impactful day for IT-security. The so called Morris Worm caused a lot of damage by abusing the gets() function of C [12]. Nowadays, this function is considered a security risk and removed from modern C standards but is still usable for backwards compatibility.

The worm targeted only specific systems (mainly servers) and worked as a reminder to system administrators that security is crucial. Large parts of the Internet had to shut down at the time.[8] This caused the foundation of a Computer Emergency Response Team, CERT for short.[12] There are a lot of CERT teams nowadays. Some are called Computer Security Incident Response Team (CSIRT).

4.2 Buffer Overflows

A buffer is a memory location that is used to store data temporarily. Buffer overflows happen when too much data is written into a buffer causing memory outside of the buffer to be rewritten and turn corrupt. The programming language C is well known for its problems concerning memory corruption and buffer overflows. Strings in C combined with specific functions are a huge security risk.

Buffer overflows were focused on the stack at first[12], as it holds the function relevant data like local variables and the return address. This means attackers can rewrite other values on the stack such as the return address enabling the attacker to redirect code execution and inject his own code into the memory behind the buffer. This means an unprotected program allows an attacker to execute his own code with the rights of the program in question.

The stack is not the only place to attack though. The heap, which is the area of memory used for dynamic memory allocation during run time, is another valid target. Heap based buffer overflows are generally different in nature to stack based ones, as it is a dynamically allocated memory space containing program data regarding the size of memory chunks that can be modified for different effects. Other possible targets are data structures like lists which are generally stored on the heap. If an attacker is able to manipulate pointers via a list, or something similar, he is able to use this in order to rewrite the return address of a function, e.g.

4.3 Programming Languages

Most of these problems only happen because of programmer error concerning memory management. It is easy to assume that picking a language like java that handles all things memory related by itself would fix most problems concerning software vulnerability. The problem is, escaping unsafe languages like C/C++ can be quite difficult, as parts of other programming languages like java are written in C or C++. [5][9]

Another important point is that many huge software projects have been growing over a lot of time. Simply changing the programming language is not an easy task because the adjusting period generally takes a lot of time and money and rewriting old code is almost impossible for bigger projects as the time required would simply not be feasible.

4.4 Countermeasures

A lot of different techniques became wildly used in order to prevent memory corruption attacks. I will go over some of the more prominent examples and their weaknesses.

4.4.1 Non-Executable Data Segments

Early buffer overflow attacks mostly reliant on overflowing the buffer through direct user input. One method, that enabled attackers to execute their on code, was achieved by overriding the return address in such a way that it points to user controlled code which is also present on the stack. In 1997 non-executable data segments where used to make the stack non-executable and hence prevent these kinds of attacks.[12]

In order to circumvent this protection the attacker would have to disable the execution prevention or find one of the many unprotected areas. E.g. the heap or libraries can be used to execute code.

4.4.2 Canaries

Canaries are values on the stack with the aim to protect from buffer overflow attacks. First proposed in 1997 they are placed, on the stack, between the return address and buffers[12][9]. The canary gets checked when the function returns. The assumption is that the change of the return address would change the canary. If the canary value got modified, the program is aborted to prevent undesirable redirection of program execution or other unwanted effects.

Canaries are partially ineffective if their location and value is known. Meaning a attacker with proper knowledge of the program and this mechanism can circumvent it rather easily if he has control over a pointer or overrides the canary with its own value.

4.4.3 ASLR

If an attacker wants to execute specific code he has to know its location in order to jump to it. Address Space Layout Randomization (ASLR) is widely used by operating systems and tries to eliminate outside code execution by randomizing the code layout. The PaX team used ASLR starting in 2001 making them one of the first. [12]

There are multiple problems with this approach. Firstly, its only probabilistic, meaning its only very likely to do something. ASLR relies on the fact that the memory is not accessible to the attacker. If an attacker is able to gain information about the memory layout he is able to adjust his attacks accordingly and bypass ASLR. Memory disclosure attacks are a common approach to circumvent ASLR.[5]

Some ASLR implementations also rely on system boots to generate new randomization. Irregular changes to the ASLR layout cause it to be predictable. ASLR is most effective if the memory regions are split as granularly as possible with large amounts of possible address space and proper randomization. If the ASLR implementation shows patterns, it is easier to circumvent. ASLR may be brute-forceable on machines with a smaller address space, even 32-bit systems.[5][6]

4.4.4 Control Flow Integrity

Control Flow Integrity (CFI) tries to limit the positions the program can jump to to only valid paths. At least according to the principal. It assumes that the code base can not be overridden (is read only). CFI is generally implemented by compilers, as most implementations of CFI are static, meaning they are generated during compile time. This causes a minimal effect on runtime. A control-flow graph is generated in order to determine possible positions for the program to jump to. The number of possible jump locations depends on the granularity of the approach. A set of possible jump locations is used for all jump locations without a fixed destination in order to restrict the flow of the program during runtime. Should the program try to jump to an invalid position, execution is aborted. The original concept uses a label for each set[1] which are then compared against the expected values on a jump. There is a general distinction between forward-edge control-flow transfers and backward-edge control-flow transfers.[7]

Forward-edge control-flow transfers are the jumps that occur through function calls or indirectly within a function due to e.g if or switch instructions.

Backward-edge control-flow transfer describes the return to a previous location via jump, making them harder to restrict properly. Forward-edge is easier to predict due to the restricted path a function will jump to, as these are all directly or indirectly stated in code. Backward-edge is a lot more complex. Many implementations don't use graphs to control backward-edges and instead use a so called shadow stack which stores prior call sites in an assumingly save memory location. [3]

It seems like a very strong deterministic defensive mechanisms and stops a huge amount of old attack types.[1] The main flaws are generally caused by bad or weaker implementations. CFI does not protect against data-only attacks as they do not alter control-flow.[5]

5 Conclusion

Finding weaknesses in software, especially unknown ones, is not a trivial task. The best way to solve software vulnerabilities is by avoiding them in the first place. This may not be possible or feasible for everything but at least for widely spread and known vulnerabilities. Knowing which vulnerabilities commonly appear in certain fields means they can be avoided in the first place. Even with modern protection against memory corruption safety is not guaranteed.

Bibliography

1: Martín Abadi, Mihai Budiu, Úlfar Erlingsson, Jay Ligatti, Control-Flow Integrity: Principles, Implementations, and Applications, 2005
2: Android , Control Flow Integrity, https://source.android.com/devices/tech/debug/cfi . Accessed 20 May 2019
3: Nathan Burow, Xinping Zhang, Mathias Payer, SoK: Shining Light on Shadow Stacks, 2019
4: Common Weakness Enumeration, 2011 CWE/SANS Top 25 Most Dangerous Software Errors, 2011, https://cwe.mitre.org/data/definitions/900.html
5: Christopher Liebchen, Advancing Memory-corruption Attacks and Defenses, 2018
6: Dr. Hector Marco-Gisbert, Dr. Ismael Ripoll-Ripoll, Exploiting Linux and PaX ASLR'sweaknesses on 32- and 64-bit systems, https://www.blackhat.com/docs/asia-16/materials/asia-16-Marco-Gisbert-Exploiting-Linux-And-PaX-ASLRS-Weaknesses-On-32-And-64-Bit-Systems-wp.pdf
7: Mathias Payer, Control-Flow Integrity: An Introduction, 2016, https://nebelwelt.net/blog/20160913-ControlFlowIntegrity.html. Accessed 20 May 2019
8: Charles Schmidt, Tom Darby, The What, Why, and How of the 1988 Internet Worm, 2001, https://snowplow.org/tom/worm/worm.html. Accessed 15 May 2019
9: Wiliam Stallings, Lawrie Brown, Computer Secruity: Principles and Practice, 2015
10: The OWASP Foundation, OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks, 2017, https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf
11: The SysSec Consortium, The Red Book:A Roadmap for Systems Security Research, 2013, http://www.syssec-project.eu/
12: The SysSec Consortium, The SysSec Consortium. Deliverable D7.1: Review of the state-of-the-art in cyberattacks, 2011, http://www.syssec-project.eu/m/page-media/3/syssec-d7.1-SoA-Cyberattacks.pdf