A

Arbitrary Code Execution

ROGER OGDEN AND BRIK ROYSTER

suh dude

CCS Concepts: •**Security and privacy** → *Software security engineering;*

General Terms: Security

Additional Key Words and Phrases: Arbitrary code execution, code injection

ACM Reference Format:
Roger Ogden and Brik Royster, 2016. Arbitrary Code Execution *ACM* V, N, Article A (January YYYY), 4 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Arbitrary code execution is a type of injection exploit in software, relating to many well-known exploit concepts, including Shellshock, email injection, and cross-site scripting. Arbitrary code execution refers to the process by which one or more processes running in a program are compromised, allowing code to be ran that wasnt intended to be. Traditionally, its assumed the goal of arbitrary code execution is a malicious one, such as the theft of information or corruption of data, but thats not always the case. There are instances where arbitrary code execution are used for research and entertainment purposes, such as finding ways to beat video games faster than would be possible under normal conditions, or to generate an entirely new experience using the games assets. In any case, arbitrary code execution is an evergreen topic because of the impact is has on the software industry, with security in software being paramount. In this paper, we will discuss several aspects of arbitrary code execution, citing and explaining examples along the way.

2. A BRIEF HISTORY OF ARBITRARY CODE EXECUTION

Arbitrary code execution has been around since assembly programming was commonplace. At its simplest form, arbitrary code execution is a software bug, not the result of a brute-force attack on a resource, such as trying millions of passwords on a particular website to steal the information of a particular user. That is, arbitrary code execution is, theoretically, preventable.

Because arbitrary code execution has been a software engineering issue for so long, it comes in many forms. Some of the most famous examples of arbitrary code execution were a result of bugs in standard C programming libraries, with strcpy being the most famous. Moreover, the fingerd daemon in Linux was known to have a vulnerability that allowed buffer overflow, and consequently, any user to execute local binaries.

Today, the web is perhaps the most common place for arbitrary code execution to occur. In 2012, the Foxypress Wordpress plugin was found to have an exploit that could allow arbitrary code execution. One of the plugins source code files, uploadify, was responsible for allowing users to upload remote files to their Wordpress servers. When the exploit was

Authors' addresses: Roger Ogden and Brik Royster, College of Engineering, Computer Science Department Boise State University

DOI: http://dx.doi.org/10.1145/0000000.0000000

found, it allowed for remote files to execute malicious PHP code on the users Wordpress server. Potential outcomes of this bug include website vandalism, database corruption, and stealing of credit card (or any other) information. It was found that versions 0.4.1.1 - 0.4.2.1 were vulnerable (1). The source code containing the exploit can be seen in Figure 1.

Furthermore, In June 2016, it was found that Google Chromes PDF viewer had the potential for arbitrary code execution. In CVE-2016-1681, found by Aleksandar Nikolic, it was discovered that if a PDF has an embedded jpeg2000 image could trigger a heap buffer overflow. The simple fix, according to Nikolic, was to change an assert statement to an if statement in the call to a OpenJPEG library function that Google used in their PDF viewer. (4)

Lastly, Apple discovered a vulnerability in configd, a DNS networking daemon that is included on devices running iOS, their mobile operating system. A heap based buffer overflow issue existed in the DNS client library. A malicious application with the ability to spoof responses from the local configd service may have been able to cause arbitrary code execution in DNS clients. This bug was reported by PanguTeam, a team of hackers responsible for many versions of the programs that allowed users to jailbreak their iOS devices. (5)

3. HOW ARBITRARY CODE EXECUTION WORKS

The possibility of all arbitrary code execution is reliant on the ability to obtain access to the instruction pointer for any particular program. The instruction pointer is a processor register that points to where the current program is executing in memory. Therefore, wherever the instruction pointer is, is whats being executed at the current time slot for that particular program. Access to the instruction pointer allows for any user to manipulate where the instruction pointer is pointing, in turn allowing for arbitrary code execution.

In Figure 2, the instruction pointer sits between L. Address and H. Address, the address space for the currently executing program. During the instruction fetch stage of the pipelined CPU process, the instruction pointer takes the instruction at that address, and loads it into the instruction register to be used for decoding and execution. The instruction pointer then increments by an amount that varies depending on what type of instruction was fetched. For example, if the instruction was add, the instruction pointer might only increment by one. However, if the instruction was a jump command, the instruction pointer might jump further down the address space, as is evident in Figure 2. Therefore, control over the instruction pointer is paramount for arbitrary code execution to occur. (2)

3.1. Buffer Overflow

The most common way to obtain access to the instruction pointer is through buffer overflow. Buffer overflow is a simple concept made possible by flaws in programs that typically run with administrator privileges. Essentially, buffer overflow occurs when the input to a program or method in a program is larger than the developer of the program expected. As a result, the method has a way of overwriting the amount of stack memory that was allocating to it, putting the program in an unknown state. When the stack memory is overwritten, the program no longer has the capability of returning to the address that called the method. As a result, the tail end of the overwritten stack could have code that points to a different address, thus compromising the instruction pointer.

The most famous example of this is the use of strcpy in the C Standard Library. The implementation of strcpy is to copy each character until it finds a 0 in the source string, which is used to denote the end of the string. This function easily allows users to overwrite the stack memory, as referenced in Figure 3. (3)

The next figure explains how to take advantage of a compromised instruction pointer. The character array called shellcode is filled with pre-written machine code that will open a shell on the compromised system. In the main method, by manipulating the value of ret initially to be pointing to the return address of main, and then manipulating that return

address to be where the contents of shellcode are stored, a user can execute the contents of shellcode.

4. HOW ARBITRARY CODE EXECUTION CAN BE PREVENTED

Because of the potentially catastrophic situations that can arise from arbitrary code execution exploits, several methods for preventing it have been implemented. Most end-all solutions take place in the C compiler (such as gcc); solutions for other languages like Bash tend to more specific to individual exploits that have occurred.

One low-level solution for preventing arbitrary code execution is the inclusion of a no-execute (NX or XD depending on the processor) bit on the CPU. In essence, this method marks certain parts of memory executable (such as the .text segment in a program), while leaving other parts of memory marked as non-executable (such as the .data segment in the program). Then, before executing any instructions, the CPU checks this no-execute bit to see if the memory it is executing from should be executed or not. If the instruction pointer ever moves to a place in memory that is not marked as executable, the program faults and exits. The inclusion and usage of a no-execute bit has been present in Windows since Windows XP SP2 (7). This method is very good in preventing code being executed from portions of memory that are controlled by the user (such as strings built from user input), but it does not stop every exploit. For instance, the Dirty COW exploit on Linux that arose recently took advantage of rewriting the .text segment, so even a no-execute bit couldnt have prevented arbitrary code from being executed.

Another solution that is in-use today by gcc is the inclusion of a stack canary during execution. The idea behind this is that most (not all) arbitrary code execution exploits take advantage of manipulating the call stack to return execution to a new place in memory and ultimately execute code that was not intended to be executed. In an attempt to prevent this, a small random integer is placed just before the call stack in memory. Since the most common way to overwrite the stack is to overflow memory writing from normal memory into the stack, any code trying to write its way into the call stack will have to overwrite the stack canary (8). This canary value is checked often, and if it is ever not equal to the original random value it was assigned, then the stack might have been manipulated and the program faults. Of course, like the no-execute bit, this solution is not full proof. In fact, the same exploit that bypasses an no-execute bit (Dirty COW) could bypass a stack canary just as easily.

Basic prevention of arbitrary code execution can stop many exploits from ever occurring, but a program is almost never truly safe. Even if a program is written and compiled with extreme care, bugs can still be exploited in the compiler or operating system. Developers should always be aware of the third-party libraries they are using in their programs and pay attention to any potential exploits that can arise from them. They should also make sure to always use the safe copying functions in C (such as strncpy instead of strcpy). Finally, they should be proactive in fixing any bugs that show signs of allowing arbitrary code execution to exist. Even if fixes are implemented and updates are launched the day an exploit is discovered, machines that arent up to date can still be compromised and huge problems can still occur.

5. CASE STUDY: ARBITRARY CODE EXECUTION IN SUPER MARIO WORLD

One great example of arbitrary code execution not being used for evil can be found in speedruns and tool-assisted play of various classic video games. In essence, a speedrun is trying to beat a video game as quickly as possible; using glitches and other unintended tricks are generally allowed and encouraged.

In several classic video games (such as Super Mario World for the SNES), glitches exist that allow arbitrary code execution to be possible. The methods for performing and taking advantages of these glitches varies from game to game, but they tend to exist more often

in older video games due to the smaller memory capacities of the retro consoles and the somewhat questionable techniques game developers used to overcome the constraints. Most cases of arbitrary code execution in video games simply result in jumping to a certain instruction in game code (e.g. the final credits) to skip large portions of the game. However, some games can be coerced into a state where inputs from the controller directly map to assembly instructions being executed by the console. This is known as total control.

To showcase an example of how powerful arbitrary code execution can be for speedruns, we will examine the methods used to beat Super Mario World in under two minutes by a human player (9).

The basis of this glitch is that the player is able to use an array in the games memory that normally stores the positions of various sprites on screen as a set of instructions to be executed that will ultimately jump game execution to the final credits. This array that stores sprite locations has a maximum size of 12 elements, and whenever a new sprite is rendered on screen, it tries to place it in the highest available index in the array. Because of this, the lower elements in the array are only ever accessed or changed when there is a large number of sprites on the screen, which makes it easy to keep them at the values the player wants. In the pursuit of efficiency, sprite positions are only ever updated when the sprite moves, and are not cleared when a sprite leaves the screen for any reason. This means sprites can be placed in a very specific location, despawned using in-game mechanics, then as long as a new sprite is not ever placed in that sprite slot, the memory still has the correct values.

Conveniently, at the very beginning of the first level, a large number of enemies appear on screen and fill up the entire sprite array (this was probably placed there by the game designers to showcase how many enemies the game can handle at once on the new hardware the game was released on). The player can very carefully despawn these enemies (by eating them and spitting them out with Yoshi) at very specific locations to write some machine code to be executed later.

Once the code has been written using sprites, the player can then execute a very complicated glitch that allows program execution to jump to the sprite arrays location in memory. The glitch involves attempting to eat a normal item with Yoshi at a specific spot in the level at the exact same time Mario jumps off of Yoshi and moves the screen to the right. When the screen moves enough to the right, it spawns another enemy in the level. However, since the item being eaten has already been despawned, the enemy spawns in the same sprite slot it was just occupying. Since Yoshi is still in the process of eating whatever item was in that sprite slot, it is able to eat the enemy that is not normally able to be eaten.

When Yoshi eats objects in Super Mario World, the game looks at a large array of memory locations mapped to each eatable object (this is called a jump table). However, since the enemy being eaten isnt supposed to be in this array in the first place, the index the game tries to access in the jump table is way out of bounds and the game attempts to jump execution to an essentially random place in memory. This location isnt actually random, but its the result of turning some other memory in the game near the jump table into an address and jumping to it. The other memory it reads happens to be the Y-position of the last particle objects drawn on the screen, which the player was also able to manipulate.

After performing this jump instruction from eating the invalid memory, the game is able to work its way into the sprite array and start executing code from there. The code written by the original sprites sets the game-state to one that tells the game to start playing the final credits, then jumps to a place in code that the game is able to recover from. Normally, eating an invalid enemy would crash the game, so execution is jumped to a subroutine intended to draw a specific boss on the screen. This subroutine causes some interesting graphical glitches, but doesnt crash the game. Once the subroutine returns, the game realizes its state is the credits and launches the proper routine to to start playing them, thus beating Super Mario World.