Engineering Resilient Systems 4/M (Part 1)

Ekku Jokinen
1703641

ABSTRACT

This report is a short high-level look on security as a permanent and constant part of the whole software development cycle and examines some of the reasons that make development teams neglect this important practice. The report uses a fictional cyber incident caused by an inside actor who has abused a software vulnerability for malicious purposes. The class of software faults that enabled the attack is briefly described as well as its popularity. After this, a potential attack scenario is described, and fuzzing is introduced as a cost-effective and straightforward secure development practice which would be able to prevent similar software failures in the future. The final section takes a look at how fuzzing can actually be used in practice, demonstrating the process of finding a buffer overflow vulnerability while fuzzing a program. After the demonstration, a recommendation is given on how the imaginary company's development team could implement the practice into their software development process.

1 Introduction

One of the most important parts of software development is implementing secure programming/development practices. By writing as much bug-free code as possible, the developer(s) minimise the amount of software bugs, or faults, that make the program susceptible to vulnerabilities that in the worst cases might lead to critical security incidents. Any vulnerability that exists in a piece of software can lead to an exploit, or software failure, which can be used by malicious users to manipulate the program to behave in an unintended way. Although being one of the most important parts of software development, security often becomes an afterthought because it requires additional budget and can add a significant amount of extra time needed to the development schedule.

Another reason it often gets payed too little attention might be that when done correctly, no "rewards" can be observed immediately other than not needing to fix major bugs or respond to major security incidents in the future. These benefits do not immediately transfer to more profit making them not as appealing especially if the development budget or time schedule is already very strict before accounting for security. In addition to this, it might be difficult to find the perfect places during development to fit security practices in. The crowdsourced security platform Bugcrowd explains the benefits of adding a bugbounty program as a part of a software development cycle and they have created a helpful graphic in illustrating how each phase of development could be paired with a security practice (Figure 1).

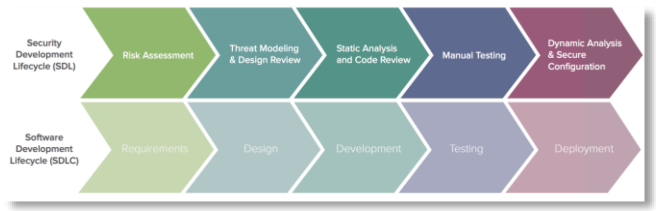

Security Development Lifecycle (SDL)
Risk Assessment
Threat Modeling & Design Review
Static Analysis and Code Review
Manual Testing
Dynamic Analysis & Secure Configuration
Software Development Lifecycle (SDLC)
Requirements
Design
Development
Testing
Deployment

Figure 1 – Matching security with software development (Bugcrowd, 2017)

2 Context

CompanyXYZ has recently had a cyber security incident where an employee gained unauthorised access to the human resources department's computer system. The company's Chief Technical Officer (CTO) contracted the author of this report to investigate the different parts of their IT systems including the network authentication system, the web frontend written in ASP.NET and the PostgreSQL database server that is used by the payroll system. The PostgreSQL payroll database was found to be the entry point for the attacker. A plausible category of security faults existing for this type of system is called memory corruption.

This vulnerability class is classified with the Common Weakness Enumeration identification number CWE-119 which is the failure to properly restrict operations within a memory buffer (The MITRE Corporation, 2020). The Mitre

Corporation keeps a list of the top 25 most dangerous software errors that are widespread and often lead to critical vulnerabilities in applications. This class of security faults was listed as the top weakness in the 2019 listing well ahead of the second most dangerous weakness by having almost twice its score (The MITRE Corporation, 2019b). Memory corruption vulnerabilities may allow a malicious user to execute arbitrary code, read restricted information or even crash a system depending on the case. CWE-119 is comprised of a broad list of related vulnerability types including the classic buffer overflow (CWE-120) and out-of-bounds writing (CWE-787).

The malicious employee seemed to have gained access or stolen a human resources employee's login credentials and after that used them to log into the salary and payroll system database. After logging in they used the recover/change password feature for the user account and abused the buffer overflow vulnerability by using a specially crafted exploit that changed the bank account details for many of the other employees. The same vulnerability could have potentially also allowed the individual to crash the server using a Denial-of-Service (DOS) type of exploit (Crunchy Data Solutions, 2019).

3 Recommendation

A very simple and effective secure development practice that would help catch and prevent memory corruption type vulnerabilities would be to add fuzzing into the development process. Fuzzing is a dynamic analysis tool/technique which involves generating and mutating data automatically and using it as input to a program (Software Assurance Forum for Excellence in Code, 2018). The goal of fuzzing is to crash the application and then see what type of data caused the crash. Fuzzing will also give an overall idea on how robust an application is and allows critical bugs to be fixed. Fuzzing is a black box type practice meaning using it does not require access to the actual source code but having access to the code will speed up the process of tracking down a potential bug and prevent the need to comb through all of the application. After finding inputs that cause a crash, they can be added to other test cases so that future builds get automatically tested for that.

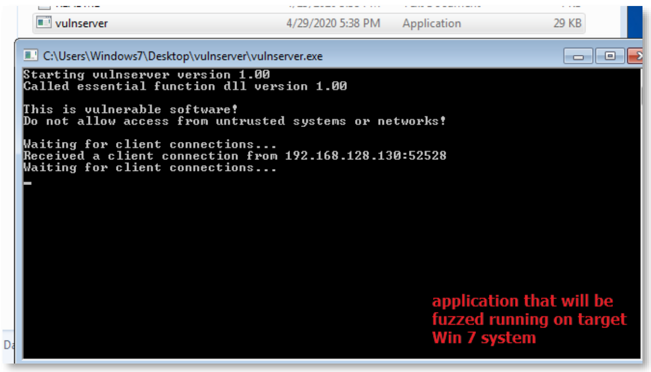
Another secure development practices worth considering would be code review. The six developers could hold weekly (or even more frequent) meetings where they analyse newly written code line by line and vigorously attempt to catch any bugs or insecure design choices. The good thing about code review is that it is an effective technique to catch (critical) issues due to many people manually going through code. However, since it would require all or many of the developers to be present, holding such meetings would take time away from actual code writing and other necessary work. Unlike fuzzing, code review cannot be reliably automated, and it can be expected to require about an hour of time for every 100 lines of code (Humphrey, 2001). Although fuzzing is not guaranteed to find any bugs and may take more time and resources if nothing is found, it makes more sense to use it as a background and test case generation practice while all of the developers are allowed to continue with their work.

A slightly simpler and less expensive secure development practice compared to fuzzing would be adding threat analysis or a security requirement review to the very beginning of the development process. This would involve the developer team discussing all the different ways a user or attacker could possibly attack the application whether it was intended or unintended. While a review like this is useful and would most likely fit into almost any development schedule, the benefits would be much less tangible than with fuzzing. It would be very unlikely that the developers could come up with the random threat scenarios that a fuzzer might find.

4 Implementation

After careful review, the PostgreSQL database was found to be using an outdated and vulnerable version of the software. This allowed the inside threat actor to abuse the buffer overflow vulnerability that existed in the password change function for an authenticated user. The Common Vulnerabilities and Exposures (CVE®) identification number for this specific vulnerability is CVE-2019-10164 (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10164).

Due to not being able to use an actual PostgreSQL database for demonstration, another service called vulnserver is used to demonstrate how fuzzing can catch critical memory corruption bugs that may lead to arbitrary code execution. The initial setup was running vulnserver on a Windows 7 Pro 64-bit virtual machine as the fuzzing target (Figure 2) and a Kali Linux 2020.2 virtual machine as the fuzzer, or attacker, system. When vulnserver is running, a network connection tool like Netcat can be used to connect to it and send multiple different types of commands (Figure 3). A debugger (Immunity Debugger) was also run on the target system to view necessary memory addresses in order to craft a working buffer overflow (not described in this report).



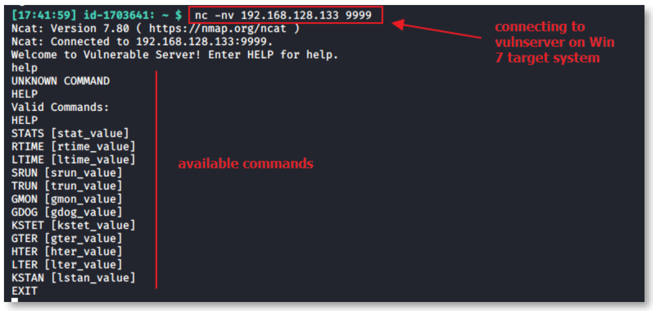
```
vulnserver    4/29/2020 5:38 PM    Application    29 KB
C:\Users\Windows7\Desktop\vulnserver\vulnserver.exe
Starting vulnserver version 1.00
Called essential function dll version 1.00

This is vulnerable software!
Do not allow access from untrusted systems or networks!

Waiting for client connections...
Received a client connection from 192.168.128.130:52528
Waiting for client connections...
_
```

application that will be fuzzed running on target Win 7 system

Figure 2 - Vulnserver running on target

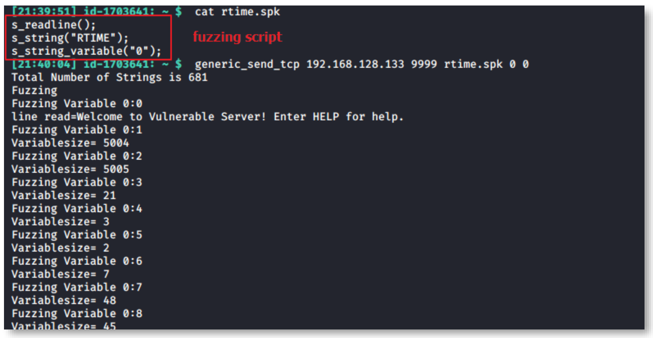


```
[17:41:59] id-1703641: ~ $  nc -nv 192.168.128.133 9999
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Connected to 192.168.128.133:9999.
Welcome to Vulnerable Server! Enter HELP for help.
help
UNKNOWN COMMAND
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
GMON [gmon_value]
GDOG [gdog_value]
KSTET [kstet_value]
GTER [gter_value]
HTER [hter_value]
LTER [lter_value]
KSTAN [lstan_value]
EXIT
```

Figure 3 - Test connection from fuzzing system (attacker)

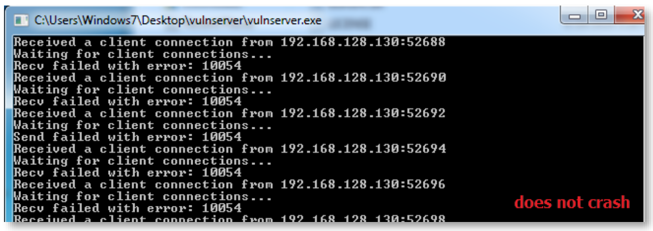
The different commands in vulnserver were fuzzed using a protocol fuzzer called SPIKE (Github, 2017). SPIKE allows the user to craft messages, or "SPIKES", that are written in a C/C++-like language and act as instructions for the fuzzer. Due to this, the fuzzer is extremely flexible and can be extended to work on almost any network-based protocol. Another way that fuzzing could be demonstrated would be to run a fuzzer straight on the system with the PostgreSQL database instead of a two-system setup. This would make the process slightly less resource-intensive and simpler due to fewer tools that need to be managed.

In the context of fuzzing vulnserver, all of the different commands were fuzzed by sending a command followed by a random amount of data in hopes of making the program behave weirdly (Figure 4). Fuzzing the 'RTIME' command did not cause a crash no matter how long the fuzzer was run (Figure 5).



```
[21:39:51] id-1703641: ~ $  cat rtime.spk
s_readline();
s_string("RTIME");
s_string_variable("0");
[21:40:04] id-1703641: ~ $  generic_send_tcp 192.168.128.133 9999 rtime.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
line read=Welcome to Vulnerable Server! Enter HELP for help.
Fuzzing Variable 0:1
Variablesize= 5004
Fuzzing Variable 0:2
Variablesize= 5005
Fuzzing Variable 0:3
Variablesize= 21
Fuzzing Variable 0:4
Variablesize= 3
Fuzzing Variable 0:5
Variablesize= 2
Fuzzing Variable 0:6
Variablesize= 7
Fuzzing Variable 0:7
Variablesize= 48
Fuzzing Variable 0:8
Variablesize= 45
```

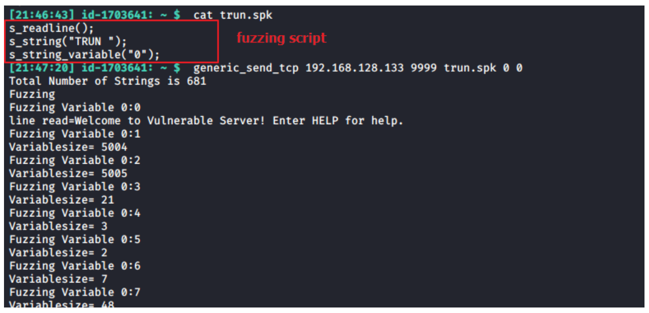
Figure 4 - Fuzzing RTIME command (no crash)



```
C:\Users\Windows7\Desktop\vulnserver\vulnserver.exe
Received a client connection from 192.168.128.130:52688
Waiting for client connections...
Recv failed with error: 10054
Received a client connection from 192.168.128.130:52690
Waiting for client connections...
Recv failed with error: 10054
Received a client connection from 192.168.128.130:52692
Waiting for client connections...
Send failed with error: 10054
Received a client connection from 192.168.128.130:52694
Waiting for client connections...
Recv failed with error: 10054
Received a client connection from 192.168.128.130:52696
Waiting for client connections...
Recv failed with error: 10054
Received a client connection from 192.168.128.130:52698
```

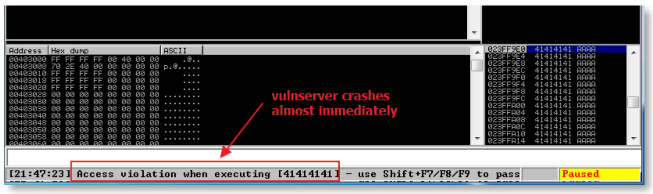
Figure 5 - vulnserver output (RTIME)

However, once the 'TRUN' command was fuzzed (Figure 6), the program crashed almost immediately, and the debugger displayed an access violation error (Figure 7). This meant that the command did not handle a certain type of input in a safe manner making the command vulnerable to a buffer overflow exploit.



```
[21:46:43] id-1703641: ~ $  cat trun.spk
s_readline();
s_string("TRUN ");
s_string_variable("0");
[21:47:20] id-1703641: ~ $  generic_send_tcp 192.168.128.133 9999 trun.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
line read=Welcome to Vulnerable Server! Enter HELP for help.
Fuzzing Variable 0:1
Variablesize= 5004
Fuzzing Variable 0:2
Variablesize= 5005
Fuzzing Variable 0:3
Variablesize= 21
Fuzzing Variable 0:4
Variablesize= 3
Fuzzing Variable 0:5
Variablesize= 2
Fuzzing Variable 0:6
Variablesize= 7
Fuzzing Variable 0:7
Variablesize= 48
```

Figure 6 - Fuzzing TRUN command (crash)



```
Address  Hex dump                  ASCII
00403000 FF FF FF FF 00 40 00 00       .@..
00403008 70 2E 40 00 00 00 00 00   p.@.....
00403010 FF FF FF FF 00 00 00 00       ....
00403018 FF FF FF FF 00 00 00 00       ....
00403020 FF FF FF FF 00 00 00 00       ....
00403028 00 00 00 00 00 00 00 00   ........
00403030 00 00 00 00 00 00 00 00   ........
00403038 00 00 00 00 00 00 00 00   ........
00403040 00 00 00 00 00 00 00 00   ........
00403048 00 00 00 00 00 00 00 00   ........
00403050 00 00 00 00 00 00 00 00   ........
00403058 00 00 00 00 00 00 00 00   ........

023FF9E0  41414141  AAAA
023FF9E4  41414141  AAAA
023FF9E8  41414141  AAAA
023FF9EC  41414141  AAAA
023FF9F0  41414141  AAAA
023FF9F4  41414141  AAAA
023FF9F8  41414141  AAAA
023FF9FC  41414141  AAAA
023FFA00  41414141  AAAA
023FFA04  41414141  AAAA
023FFA08  41414141  AAAA
023FFA0C  41414141  AAAA
023FFA10  41414141  AAAA
023FFA14  41414141  AAAA

[21:47:23] Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass    Paused
```

Figure 7 - Debugger catching crash (TRUN)

Explaining the process of crafting the final exploit in detail is outside the scope of this report but the next step was to analyse the specific input that caused the crash. After this, using memory addresses found with the debugger an exploit with a payload was crafted, which for testing purposes was to simply open calculator.exe remotely. Once the final exploit was sent as a "command" to vulnserver, the calculator was successfully opened proving that arbitrary code could be executed (Figure 8).

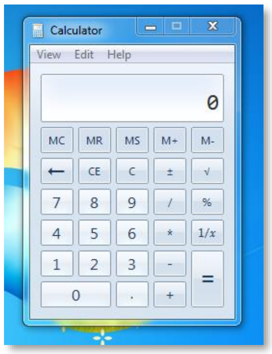


Figure 8 - Opening up calculator with exploit

A straightforward way to implement fuzzing into the development process would be to choose one developer who would oversee it. This person would then design new types of fuzzing tests and check on the results for example in the morning and at the end of the workday (assuming the fuzzer runs constantly). In the event of a crash, they would investigate it further and eventually add that type of input as an automated test case along with any other build testing the team uses. Implementing the fuzzing process for the PostgreSQL database software would be similar in nature to the process of fuzzing vulnserver. There are also many alternative fuzzers available both open source and commercial in case SPIKE does not work.

References

Bugcrowd (2017) *How does a bug bounty fit into my SDLC?* Available at: https://www.bugcrowd.com/blog/how-does-a-bug-bounty-fit-into-my-sdlc/ (Accessed: 29 April 2020).

Crunchy Data Solutions (2019) *Explaining CVE-2019-10164 + PostgreSQL security best practices*. Available at: https://info.crunchydata.com/blog/explaining-cve-2019-10164-with-postgresql-security-best-practices (Accessed: 28 April 2020).

F-Secure (2020) *15 minute guide to fuzzing*. Available at: https://www.f-secure.com/en/consulting/our-thinking/15-minute-guide-to-fuzzing (Accessed: 28 April 2020).

Github (2017) *Guilhermeferreira / spikepp*. Available at: https://github.com/guilhermeferreira/spikepp (Accessed: 29 April 2020).

Humphrey, W. S. (2001) *Winning with software: an executive strategy*. Boston: Pearson Education

Sampson, A. (2020) 'Introduction to secure software development' [PowerPoint presentation]. CMP417: Engineering sesilient systems. Abertay University. Available at: https://mylearningspace.abertay.ac.uk/ (Accessed: 29 April 2020).

Software Assurance Forum for Excellence in Code (2018) *Fundamental practices for secure software development*. Available at: https://safecode.org/wp-content/uploads/2018/03/SAFECode_Fundamental_Practices_for_Secure_Software_Development_March_2018.pdf (Accessed: 27 April 2020).

The MITRE Corporation (2019a) *CVE-2019-10164*. Available at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10164 (Accessed: 27 April 2020).

The MITRE Corporation (2019b) *2019 CWE top 25 most dangerous software errors*. Available at: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html (Accessed: 27 April 2020).

The MITRE Corporation (2020) *CWE-119: improper restriction of operations within the bounds of a memory buffer*. Available at: https://cwe.mitre.org/data/definitions/119.html (Accessed: 27 April 2020).