CSAO Assessment One

Table of Contents

Table of Figures

Section A

Part I

A stack buffer overflow occurs when the program attempts to write to a memory address on the stack outside of the intended buffer bounds. This happens when a program writes more data to the buffer on the stack than what it is allocated for the buffer. An attacker could take advantage of this and as a result they can make the program do what they want. As examples, they can run shellcode to gain access a shell or they can try to access a memory address to gain information. Modern computer systems have protection systems against these attacks, however the architecture determines the extent of this security. x86 computers only have around 16 bits available for memory addresses, that is 2^{16} = 65536 addresses available. x86_64 computers have around or more than 32 bits, so 2^{32} = 4,294,967,296 addresses. (MITRE Organisation, 2021; Oracle Corporation, 2010)

One of the protection systems is Address Space Layout Randomization (ASLR), whose effectiveness increases with the amount of bits and memory addresses available.

ASLR is a widely used protection method that randomizes memory addresses of processes to try to deter exploitation methods that rely in knowing the exact location of the process. ASLR takes the program's code, stack and heap and it places them at random addresses in the program's memory address space. The entropy of the ASLR system determines how randomized the memory addresses are and, as a result, how secure and resistant it is to brute forcing. (Marco-Gisbert & Ripoll Ripoll, 2019)

The Linux PaX project invented the term "ASLR". The team published the first design of ASLR in July 2001 as a patch for the Linux kernel. Linux has then integrated built-in support for ASLR in 2005. (Dang, 2009)

PaX ASLR had some holdbacks. At the time, 32-bit systems were the most common. 32-bit systems had a low entropy of 16 bits and ASLR only randomized the addresses at program loading and never after that. A brute force attack on the memory could be done in a matter of minutes and it would likely go unnoticed. A 64-bit system would likely have more than 32 bits of entropy, even with large page sizes, therefore a brute force attack is most certainly not going to succeed and an attempt at it of this magnitude is probably going to be noticed. (Shacham, et al., 2007)

In conclusion, ASLR is much more effective on 64-bit systems because, even with large pages at 4096KB, this still allows for 42 bits of entropy, which is approximately 4 trillion addresses. (Detter & Mutschlechner, 2015; Shacham, et al., 2007)

Part II

Modern systems have included many ways of preventing buffer overflows. These systems have been introduced to ensure that malicious code wasn't executed from the stack and preventing the program from becoming a serious security vulnerability.

One of these systems are canaries. Canaries are known values placed before critical stack values, like the return pointer, to protect against buffer overflow attacks. Canaries are added during compilation and in case of a buffer overflow, they are usually overwritten first. In the event of a corrupted canary, a failed verification would occur – the verifications can be placed, for example, right before a return command –

and the program would alert of an overflow. The alert can be handled by, for instance, by invalidating the corrupted data or by simply halting the program. (SANS Institute, 2021)

There are multiple types of canaries. The simplest form would be the NULL canary. In a 32-bit system, it would place 4 NULL bytes (0x00) just before the stack frame pointer (ebp) and the return pointer. This is a very predictable value, so it provides little protection. Another type is a terminator canary. This would also include the null terminator, but it would also include the carriage return, CR or 0x0d, the line feed, LF or 0x0a, and the form feed, FF or 0xff. Regardless, even with the new terminator additions, an attacker can still be able to overwrite the canary and bypass it as the values are known and predictable. (SANS Institute, 2021)

A type of canary that offers better protection would be the random canary. Random canaries are randomly generated but they usually consist of a NULL byte followed by three random bytes – this has 24 bits of entropy, whereas a 64-bit canary would have an extra 32 bits of entropy. The NULL byte would try to halt string operations while the random bytes would try to obfuscate the canary so that it is less obvious to the attacker. The randomization is usually done by an entropy-gathering daemon, such as EGD. The random XOR canary is like the random canary, however the canary is XOR-scrambled against a dynamic value, usually the base pointer, ebp. As a result, the attacker would need the canary, the algorithm and the dynamic value in order to re-generate the original canary needed to bypass the protection. (Cowan, et al., 1998, p. 5; Hawkins, et al., 2016, pp. 1-2; SANS Institute, 2021)

Another type of protection against a buffer overflow attack is by enforcing a policy on the stack memory region that disables all execution from the stack. This is also known as executable space protection and it is uses the “NX bit”, No-eXecute bit, that is available as a hardware feature in processors. Executable stack protection marks regions of memory as non-executable. It prevents the execution of arbitrary code in the stack and the heap via a buffer overflow. Any attempt to insert malicious code will cause an exception to occur. (Yadav, et al., 2011)

Part III

The Advanced Encryption Standard (AES) is a symmetric block cipher that processes blocks of data of 128 bits using keys with lengths of 128, 192 and 256 bits. The AES algorithm uses the same key to encrypt and decrypt information. A side-channel attack via the CPU cache could break a system’s AES encryption in the cloud as cloud computing’s advantages are brought in by resource sharing methods such as co-location and data deduplication. In cloud computing, different users run their virtual machines on the same physical machine separated by a virtualization layer implemented by the virtual machine manager and supervised by a hypervisor. The Worcester Polytechnic Institute demonstrates this vulnerability by mounting cross-VM Flush+Reload cache attacks on VMs to recover the keys of an AES implementation of OpenSSL v1.0.1 running inside the victim VM. This modified flush+reload attack only takes seconds to minutes to succeed in a cross-VM setting, without the need of long-term co-location, which is needed by other fine grain attacks. A cache side-channel attack doesn’t interrupt the ongoing cryptographic operation and it is invisible to the victim. Some recent examples of such an attack would be Meltdown and Spectre. (Irazoqui, et al., 2014; Kocher, et al., 2020; Lipp, et al., 2020; NIST Computer Security Resource Center, 2001)

Processes running on the same processor share processor caches. Caches are multilayer structures of fast memory that sit between the CPU cores and the RAM. A CPU usually has three layers of cache, L1 cache, L2 cache and L3 cache. There are differences in speed and capacity between the layers. L1 cache provides the highest reading and writing speeds while the capacity is quite low. L3 cache, also known as the Last-Level Cache, or LLC, is the opposite, having a slower, but still faster than memory, reading and writing speeds but with the largest capacity out of the cache levels. (Irazoqui, et al., 2014; Su & Zeng, 2021; Yarom & Falkner, 2014)

The flush+reload method attacks the L3 cache. It aims to exploit a weakness in Intel X86 processors where memory pages are shared between non-trusting processes because page sharing exposes processes to information leaks. This is done as the system software wants to reduce the memory footprint of a system therefore it shares identical memory pages between processes on the system. Physical mechanisms on the hardware are used to maintain isolation between the non-trusting processes such as enforce read only and copy-on-write semantics for shared pages. The University of Adelaide tested the efficacy of the flash+reload attack by utilizing it to extract the private encryption keys from a victim process running GnuPG 1.4.13. The flush+reload attack was tested on two unrelated processes in a single operating system and between processes running in separate virtual machines. The attack can recover 96.7% of the bits of the secret key just by observing a single signature or decryption round. (GnuPG Project, 2021; Yarom & Falkner, 2014)

The flush+reload attack uses a spy process to ascertain or not if specific cache lines have been accessed or not by the victim code. In the flushing stage, the desired memory lines are flushed from the entirety of the cache hence making sure that they will have to be retrieved from the main memory next time they need to be accessed. In the target accessing stage, the attacker waits until the victim runs a fragment of the code which might use the flushed memory lines. In the reloading stage, the previously flushed memory lines are reloaded and the attacker measures the time taken to reload them. Depending on the reloading time, the attacker decides whether the victim accessed the memory line and if so, the line would be present in the cache. Otherwise, the victim did not access the corresponding memory line and it wouldn't be present in the cache. The timing difference between a cache hit and a cache miss makes the access easily detectable by the attacker. This is how side-channel attacks via the CPU cache affect the hardware security. (Irazoqui, et al., 2014; Su & Zeng, 2021; Yarom & Falkner, 2014)

Section B

Part I

Firstly, just making the "Binary_X" file executable and running it without arguments, it displays some ASCII art and how to run the file.

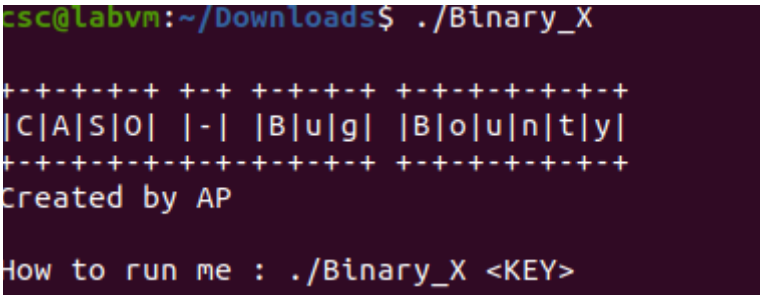

Figure B-1 - Running the binary with no arguments

Running the file with an argument, in this case “test”, it outputs that the key was incorrect and that the access is denied. This informs me that at some point in the code, the argument inputted and a stored string get compared.

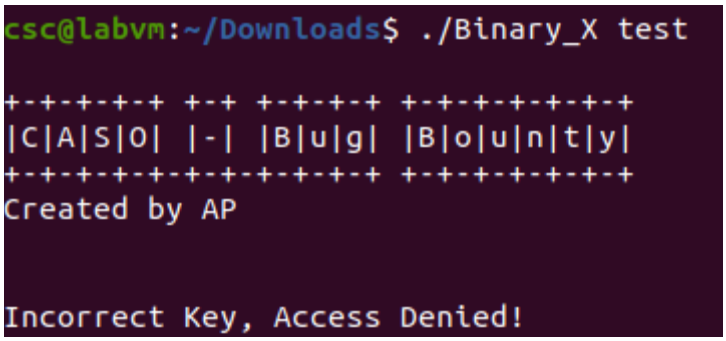

Figure B-2 - Running the binary with a "test" argument

I ran the binary file in GDB and, when disassembling the main function, we can tell that the first few lines that use the puts library are related to the ASCII art.

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000555555555189 <+0>:     endbr64
   0x000055555555518d <+4>:     push   rbp
   0x000055555555518e <+5>:     mov    rbp,rsp
   0x0000555555555191 <+8>:     sub    rsp,0x40
   0x0000555555555195 <+12>:    mov    DWORD PTR [rbp-0x34],edi
   0x0000555555555198 <+15>:    mov    QWORD PTR [rbp-0x40],rsi
   0x000055555555519c <+19>:    cmp    DWORD PTR [rbp-0x34],0x2
   0x00005555555551a0 <+23>:    je     0x5555555551e8 <main+95>
   0x00005555555551a2 <+25>:    lea    rdi,[rip+0xe5f]        # 0x555555556008
   0x00005555555551a9 <+32>:    call   0x555555555070 <puts@plt>
   0x00005555555551ae <+37>:    lea    rdi,[rip+0xe7b]        # 0x555555556030
   0x00005555555551b5 <+44>:    call   0x555555555070 <puts@plt>
   0x00005555555551ba <+49>:    lea    rdi,[rip+0xe97]        # 0x555555556058
   0x00005555555551c1 <+56>:    call   0x555555555070 <puts@plt>
   0x00005555555551c6 <+61>:    lea    rdi,[rip+0xeaf]        # 0x55555555607c
   0x00005555555551cd <+68>:    call   0x555555555070 <puts@plt>
   0x00005555555551d2 <+73>:    lea    rdi,[rip+0xeb7]        # 0x555555556090
   0x00005555555551d9 <+80>:    call   0x555555555070 <puts@plt>
   0x00005555555551de <+85>:    mov    edi,0x0
   0x00005555555551e3 <+90>:    call   0x555555555090 <exit@plt>
   0x00005555555551e8 <+95>:    lea    rdi,[rip+0xe19]        # 0x555555556008
   0x00005555555551ef <+102>:   call   0x555555555070 <puts@plt>
   0x00005555555551f4 <+107>:   lea    rdi,[rip+0xe35]        # 0x555555556030
   0x00005555555551fb <+114>:   call   0x555555555070 <puts@plt>
   0x0000555555555200 <+119>:   lea    rdi,[rip+0xe51]        # 0x555555556058
   0x0000555555555207 <+126>:   call   0x555555555070 <puts@plt>
   0x000055555555520c <+131>:   lea    rdi,[rip+0xe69]        # 0x55555555607c
   0x0000555555555213 <+138>:   call   0x555555555070 <puts@plt>
```

Figure B-3 - The disassembled main function

```
   0x5555555551d9 <main+80>:    call   0x555555555070 <puts@plt>
   0x5555555551de <main+85>:    mov    edi,0x0
   0x5555555551e3 <main+90>:    call   0x555555555090 <exit@plt>
=> 0x5555555551e8 <main+95>:    lea    rdi,[rip+0xe19]        # 0x555555556008
   0x5555555551ef <main+102>:   call   0x555555555070 <puts@plt>
   0x5555555551f4 <main+107>:   lea    rdi,[rip+0xe35]        # 0x555555556030
   0x5555555551fb <main+114>:   call   0x555555555070 <puts@plt>
   0x555555555200 <main+119>:   lea    rdi,[rip+0xe51]        # 0x555555556058
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffdee0 --> 0x7fffffffe018 --> 0x7fffffffe345 ("/home/csc/Downloads
0008| 0x7fffffffdee8 --> 0x2555552fd
0016| 0x7fffffffdef0 --> 0x7ffff7fb1fc8 --> 0x0
0024| 0x7fffffffdef8 --> 0x5555555552b0 (<__libc_csu_init>:     endbr64)
0032| 0x7fffffffdf00 --> 0x0
0040| 0x7fffffffdf08 --> 0x5555555550a0 (<_start>:      endbr64)
0048| 0x7fffffffdf10 --> 0x7fffffffe010 --> 0x2
0056| 0x7fffffffdf18 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x00005555555551e8 in main ()
gdb-peda$ x/1s 0x555555556008
0x555555556008: "\n+-+-+-+-+ +-+ +-+-+-+ +-+-+-+-+-+-+"
gdb-peda$ x/1s 0x555555556030
0x555555556030: "|C|A|S|O| |-| |B|u|g| |B|o|u|n|t|y|"
gdb-peda$
```

Figure B-4 - The first few lines are related to the ASCII art

After the ASCII art was loaded, some strings that look like potential keys to unlock the program are getting loaded on the stack.

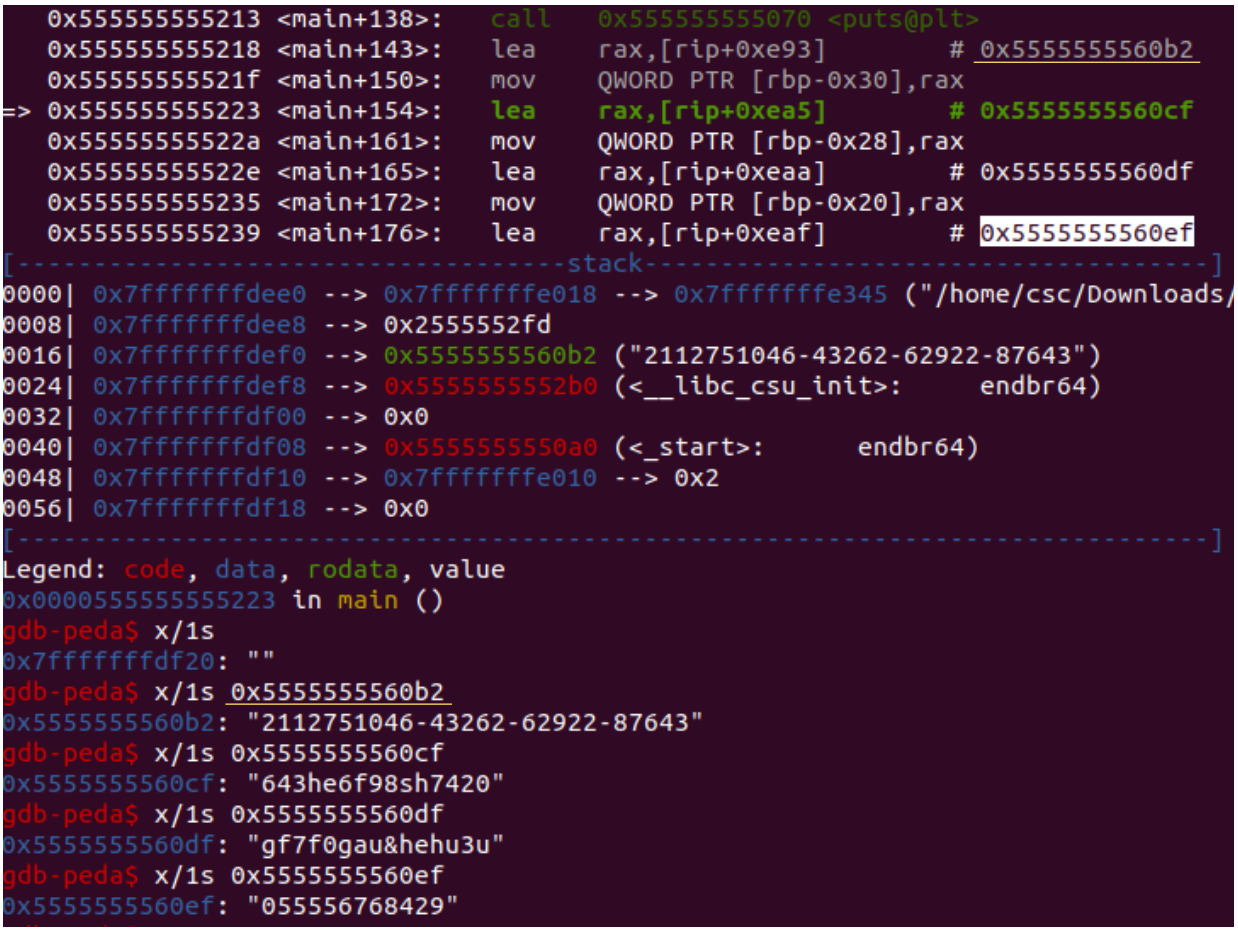

Figure B-5 - Potential keys

When approaching the part where the strings are compared, the strcmp function, the test argument inputted is loaded into RAX and another code is loaded into RDX and RSI. This could be the potential key.

Figure B-6 - The "test" argument being loaded into RAX as a potential key is loaded into RDX and RSI

When calling strcmp, GDB is giving a helping hand and it shows us the values compared. This gives more solid evidence that “055556768429” is in fact the key.

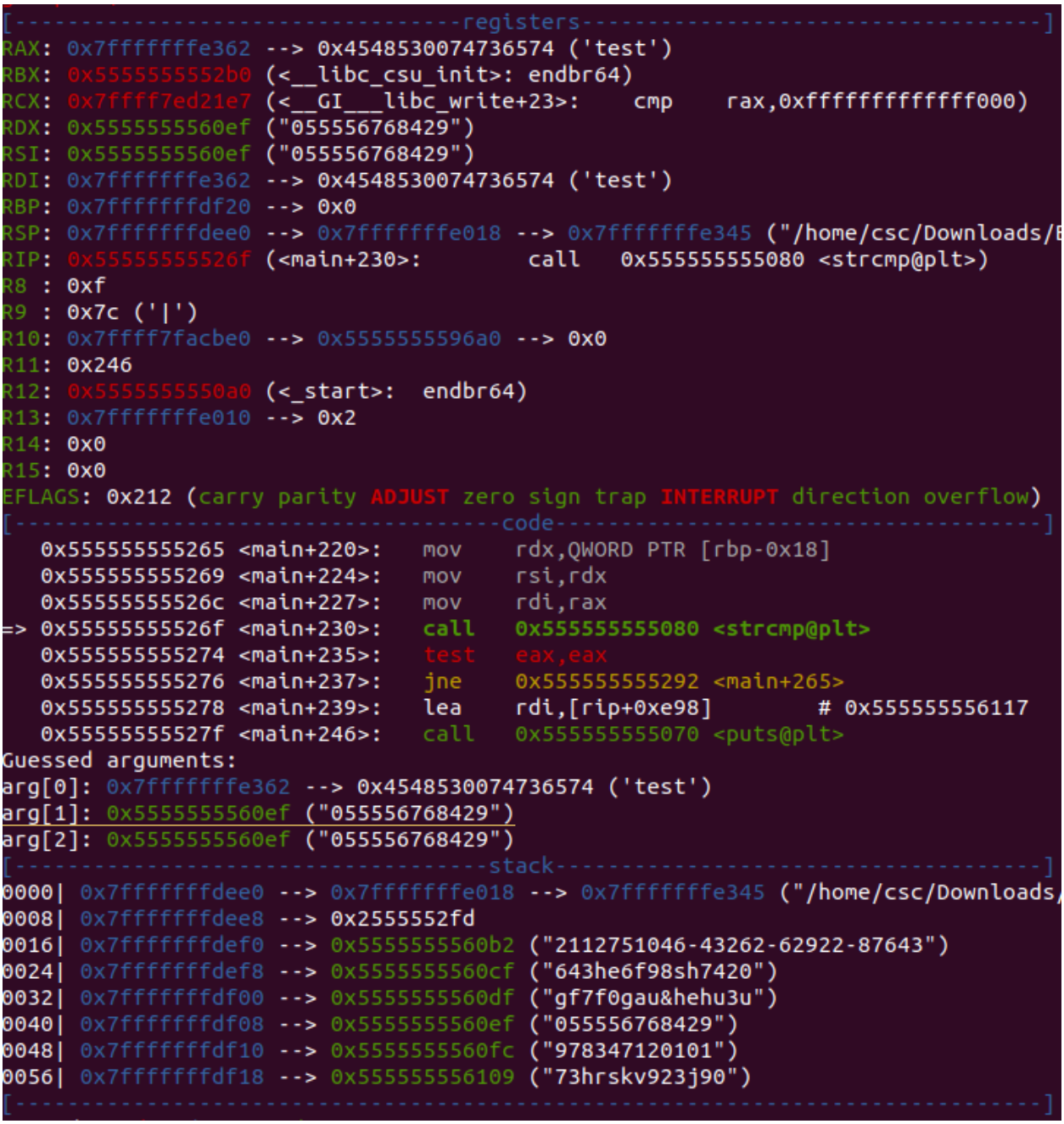

Figure B-7 - The arguments for "strcmp" are shown

When reaching the “jne” instruction line, a jump is taken. It loads “"\nIncorrect Key, Access Denied!" into rdi. Continuing the program, it outputs that “test” key is incorrect.

```
   0x55555555526c <main+227>:    mov    rdi,rax
   0x55555555526f <main+230>:    call   0x555555555080 <strcmp@plt>
   0x555555555274 <main+235>:    test   eax,eax
=> 0x555555555276 <main+237>:    jne    0x555555555292 <main+265>
 | 0x555555555278 <main+239>:    lea    rdi,[rip+0xe98]        # 0x555555556117
 | 0x55555555527f <main+246>:    call   0x555555555070 <puts@plt>
 | 0x555555555284 <main+251>:    lea    rdi,[rip+0xe9c]        # 0x555555556127
 | 0x55555555528b <main+258>:    call   0x555555555070 <puts@plt>
 |->    0x555555555292 <main+265>:       lea    rdi,[rip+0xeaf]        # 0x555555556148
       0x555555555299 <main+272>:       call   0x555555555070 <puts@plt>
       0x55555555529e <main+277>:       mov    eax,0x0
       0x5555555552a3 <main+282>:       leave
                                                                  JUMP is taken
```

Figure B-8 - The branching jump is taken

```
RAX: 0x44 ('D')
RBX: 0x5555555552b0 (<__libc_csu_init>: endbr64)
RCX: 0xffffffff
RDX: 0x30 ('0')
RSI: 0x5555555560ef ("055556768429")
RDI: 0x555555556148 ("\nIncorrect Key, Access Denied!")
RBP: 0x7fffffffdf20 --> 0x0
RSP: 0x7fffffffdee0 --> 0x7fffffffe018 --> 0x7fffffffe345 ("/home/csc/Downloads/B
RIP: 0x555555555299 (<main+272>:        call   0x555555555070 <puts@plt>)
R8 : 0xf
R9 : 0x7c ('|')
R10: 0x7ffff7facbe0 --> 0x5555555596a0 --> 0x0
R11: 0x246
R12: 0x5555555550a0 (<_start>:  endbr64)
R13: 0x7fffffffe010 --> 0x2
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x55555555528b <main+258>:    call   0x555555555070 <puts@plt>
   0x555555555290 <main+263>:    jmp    0x55555555529e <main+277>
   0x555555555292 <main+265>:    lea    rdi,[rip+0xeaf]        # 0x555555556148
=> 0x555555555299 <main+272>:    call   0x555555555070 <puts@plt>
   0x55555555529e <main+277>:    mov    eax,0x0
   0x5555555552a3 <main+282>:    leave
   0x5555555552a4 <main+283>:    ret
   0x5555555552a5:      nop    WORD PTR cs:[rax+rax*1+0x0]
Guessed arguments:
arg[0]: 0x555555556148 ("\nIncorrect Key, Access Denied!")
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffdee0 --> 0x7fffffffe018 --> 0x7fffffffe345 ("/home/csc/Downloads/
0008| 0x7fffffffdee8 --> 0x2555552fd
0016| 0x7fffffffdef0 --> 0x5555555560b2 ("2112751046-43262-62922-87643")
0024| 0x7fffffffdef8 --> 0x5555555560cf ("643he6f98sh7420")
0032| 0x7fffffffdf00 --> 0x5555555560df ("gf7f0gau&hehu3u")
0040| 0x7fffffffdf08 --> 0x5555555560ef ("055556768429")
0048| 0x7fffffffdf10 --> 0x5555555560fc ("978347120101")
0056| 0x7fffffffdf18 --> 0x555555556109 ("73hrskv923j90")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x000055555555299 in main ()
gdb-peda$ x/1s 0x555555556148
0x555555556148: "\nIncorrect Key, Access Denied!"
```

Figure B-9 - The "test" key is incorrect

Running the program again with the argument as “055556768429”, the general behaviour of the program does not change until the program calls strcmp.

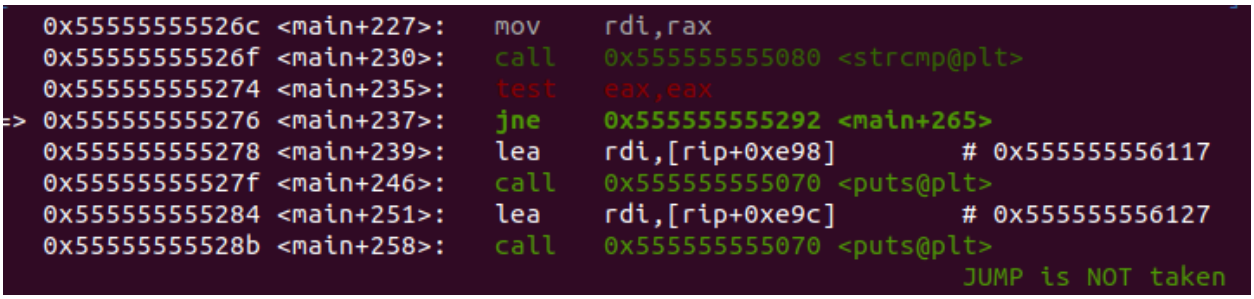

Figure B-10 - Inputting the suspected key, the branching jump is not taken

This time around, the jump is not taken. Continuing the code outputs “access granted”, confirming that the key was correct.

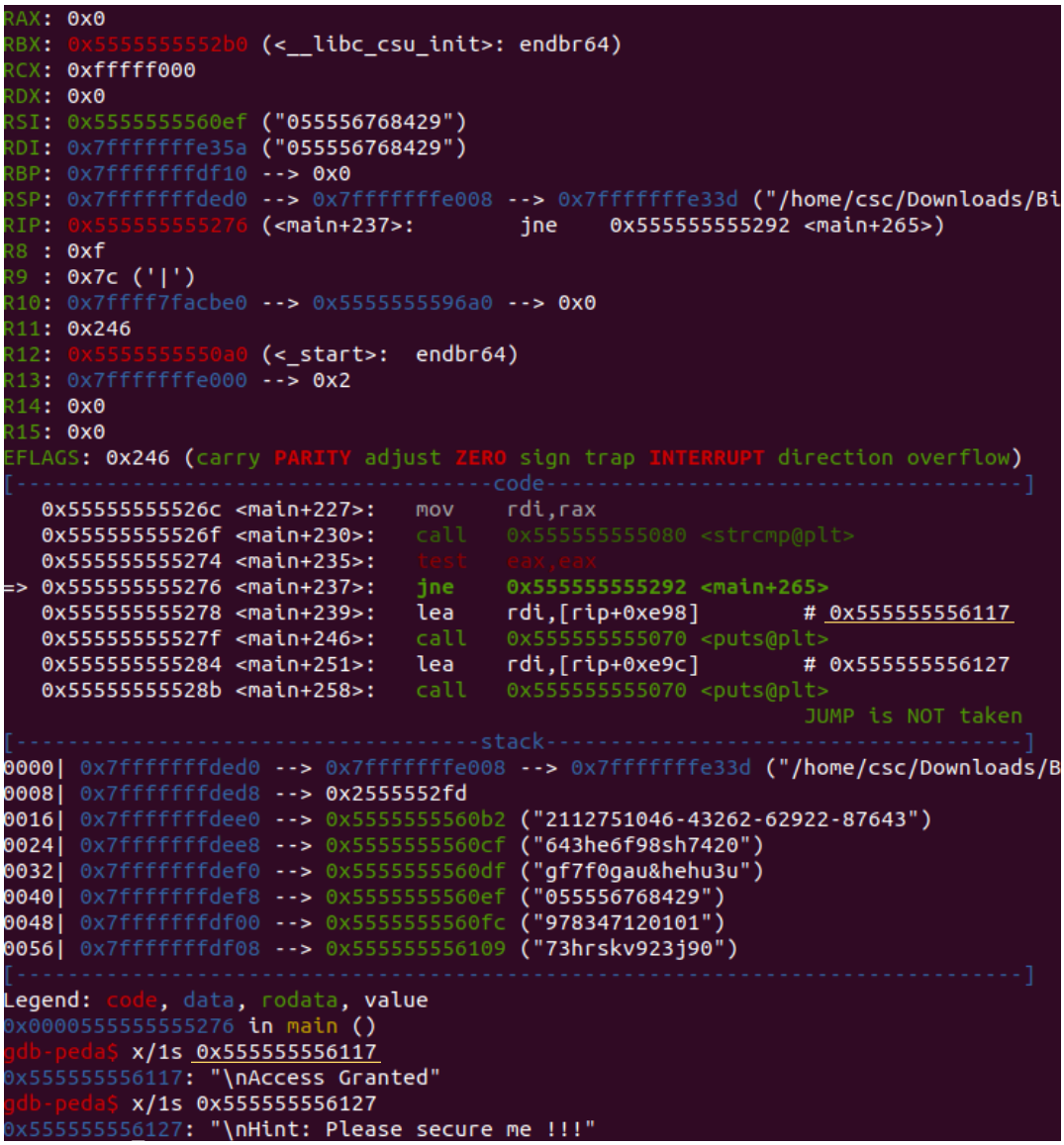

Figure B-11 - "Access Granted" is stored in a memory address

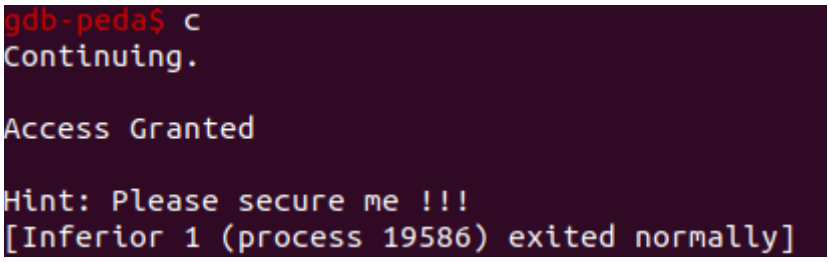

Figure B-12 - Access Granted

There is another method to bypass the passphrase validation and get this message. It involves the “test eax,eax” instruction, the jne instruction and how they are related. The test instruction performs a

bitwise AND on two operands, in this case eax and eax. The jne instruction only takes a jump if the zero flag is set to 0. In the code, the jne command at <main+237> , if the jump is taken (in case of a wrong passphrase), it would skip over the “Access Granted”.

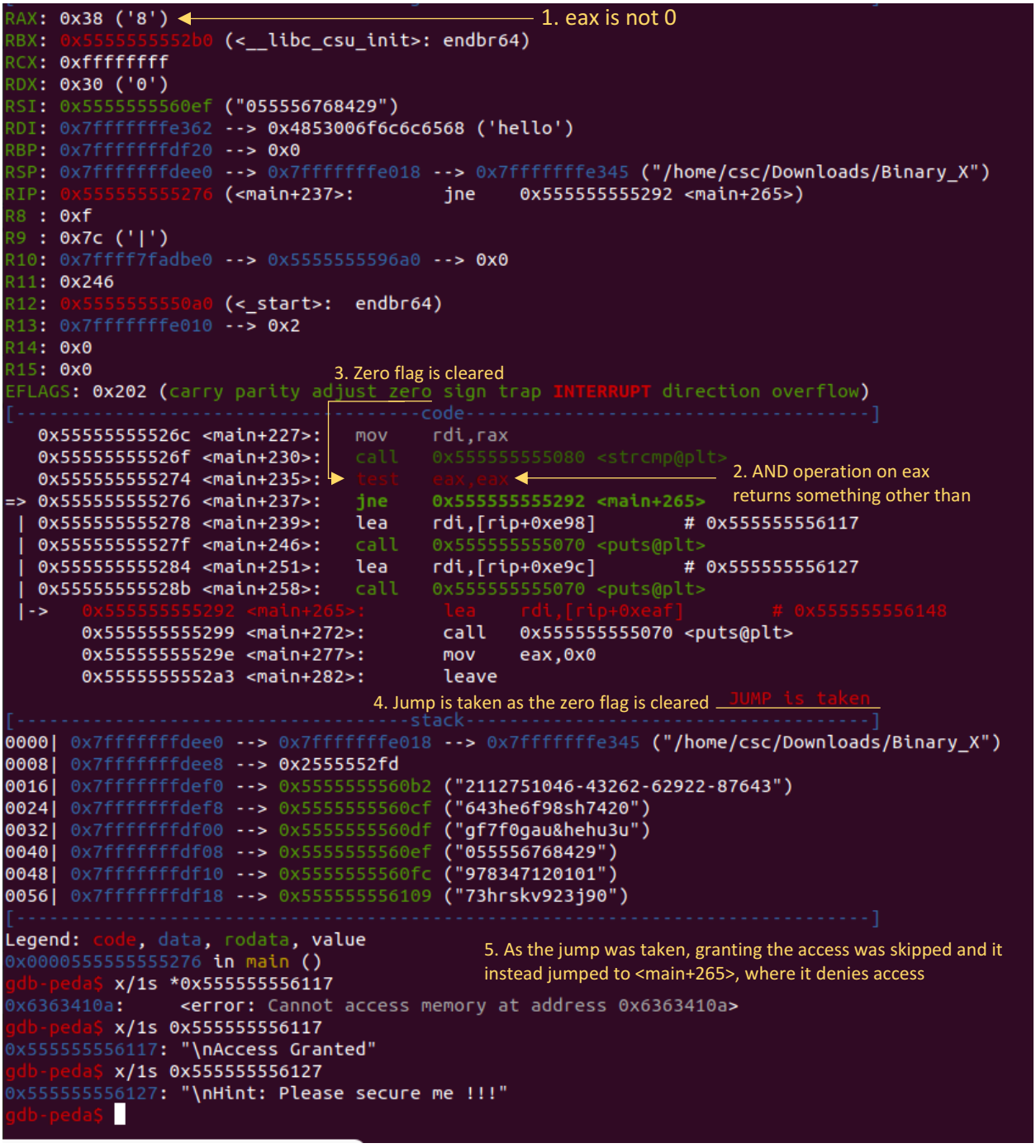

Figure B-13 - Zero Flag cleared, access denied

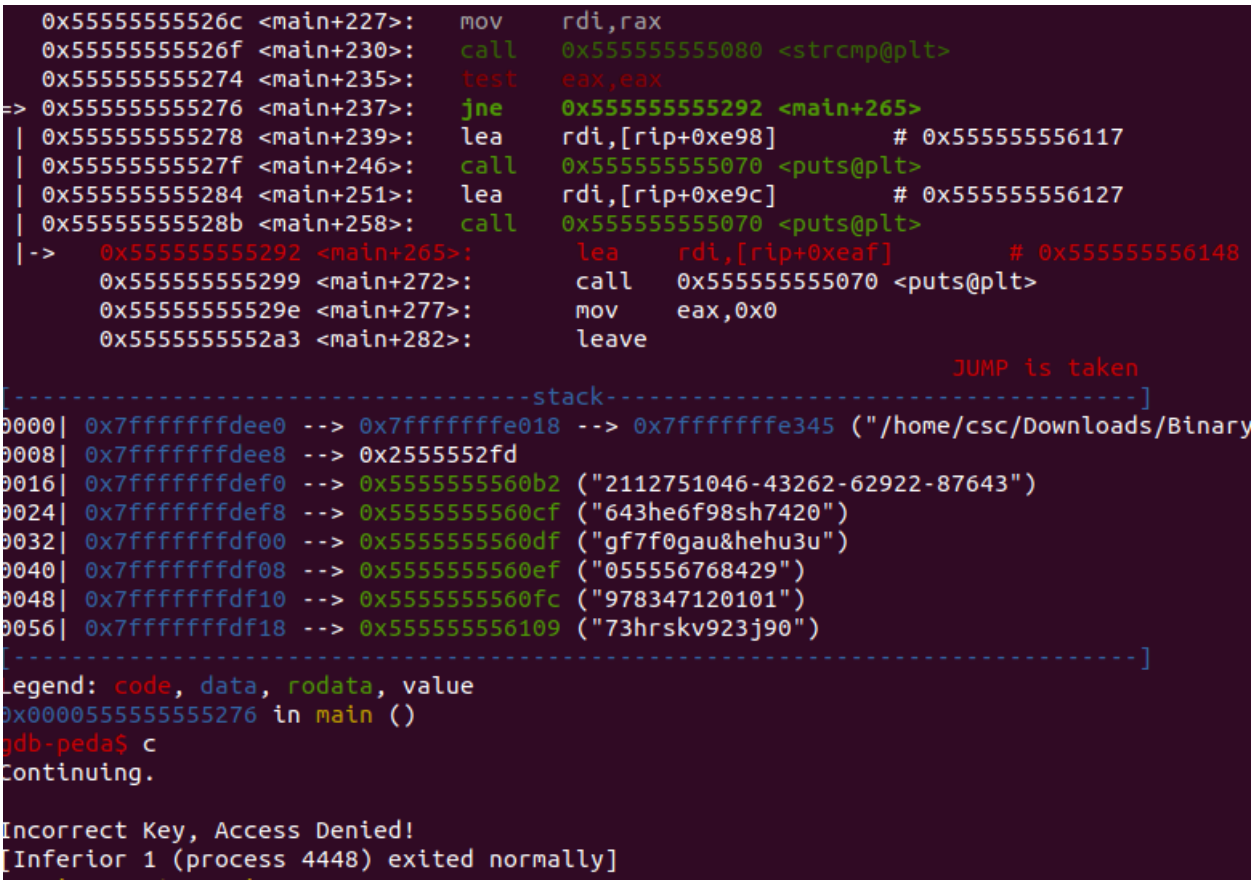

Figure B-14 - Access denied

Placing a breakpoint just before the “test eax,eax” instruction, I can run the binary again with any input. Right on the “test” line, I will set RAX to 0. This should bypass the passphrase validation and it should grant me access.

Figure B-15 - Setting RAX to 0

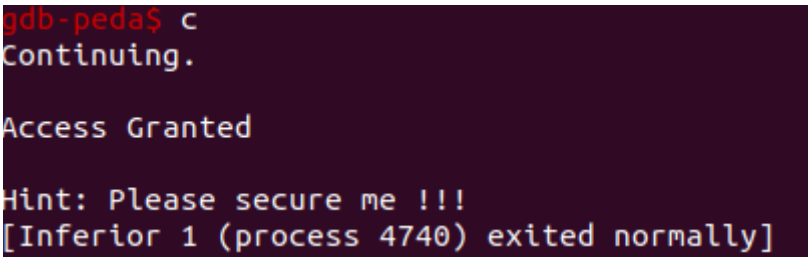

Figure B-16 - Continuing after setting RAX to 0

Continuing after setting RAX to 0 has successfully bypassed the passphrase validation.

Part II

A major flaw in the program is that the passphrase is stored in plaintext. Passwords or passphrases shouldn't be stored in plaintext. They are exponentially more secure being stored as hashes as there is no need to compare the plaintext passwords and placing other accounts of the user at risk. With a hash, the program is checking to see if the user knows the password and not actually checking the password, like the binary file does. We can do that by comparing hashes. Implementing this would take the user's input and immediately hashing it, then comparing the hashed input with the stored hash and seeing if they are the same, meaning that the user does know the password. If an attacker found the hash, it wouldn't be useful as a hash is not reversible and inputting the hash would run the hash algorithm on the input a second time, therefore giving a different output.

Unfortunately, changing the registers during runtime cannot be mitigated. A way to try to hinder changing a register would be to protect the code through anti-debugging. Once a process detects the debugger, it must penalize it. A penalty can be simply terminating the process. However, this lets the attacker know the anti-debugging code's location and the attacker could just run it again and try to bypass the anti-debugging. A more useful anti-debugging technique would be to subtly avoid the checking algorithm in the Binary_X file, or to change it with a simpler algorithm, or to introduce a fault that causes the code to crash during the execution. (Lesk, et al., 2007)

References

Cowan, C. et al., 1998. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.* [Online]
Available at:
https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf
[Accessed 17 February 2022].

Dang, A., 2009. *Behind Pwn2Own: Exclusive Interview With Charlie Miller.* [Online]
Available at: https://www.tomshardware.com/reviews/pwn2own-mac-hack,2254-4.html
[Accessed 16 February 2022].

Detter, J. & Mutschlechner, R., 2015. *Performance and Entropy of Various ASLR Implementations.* [Online]

Available at: https://pages.cs.wisc.edu/~riccardo/736finalpaper.pdf
[Accessed 16 February 2022].

GnuPG Project, 2021. *GnuPG Frequently Asked Questions.* [Online]
Available at: https://www.gnupg.org/faq/gnupg-faq.html#compatible
[Accessed 02 March 2022].

Hawkins, W. H., Hiser, J. D. & Davidson, J. W., 2016. *Dynamic Canary Randomization for Improved Software Security.* [Online]
Available at: https://dl.acm.org/doi/pdf/10.1145/2897795.2897803
[Accessed 17 February 2022].

Irazoqui, G., Inci Sisan, M., Eisenbarth, T. & Sunar, B., 2014. *Wait a minute! A fast, Cross-VM attack on AES.* [Online]
Available at: https://eprint.iacr.org/2014/435.pdf
[Accessed 1 March 2022].

Kocher, P. et al., 2020. *Spectre Attacks: Exploiting Speculative Execution.* [Online]
Available at: https://spectreattack.com/spectre.pdf
[Accessed 27 February 2022].

Lesk, M., Stytz, M. R. & Trope, R. L., 2007. *Software Protection through Anti-Debugging.* [Online]
Available at: https://ieeexplore.ieee.org/abstract/document/4218560
[Accessed 06 March 2022].

Lipp, M. et al., 2020. *Meltdown: Reading Kernel Memory from User Space.* [Online]
Available at: https://meltdownattack.com/meltdown.pdf
[Accessed 27 February 2022].

Marco-Gisbert, H. & Ripoll Ripoll, I., 2019. *Address Space Layout Randomization Next Generation.* [Online]
Available at: https://www.mdpi.com/2076-3417/9/14/2928/htm
[Accessed 16 February 2022].

MITRE Organisation, 2021. *CWE-121: Stack-based Buffer Overflow.* [Online]
Available at: https://cwe.mitre.org/data/definitions/121.html
[Accessed 16 February 2022].

NIST Computer Security Resource Center, 2001. *FIPS 197, Advanced Encryption Standard (AES).* [Online]
Available at: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf
[Accessed 1 March 2022].

Oracle Corporation, 2010. *Using the 64-bit Architecture.* [Online]
Available at: https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032f/index.html
[Accessed 16 February 2022].

SANS Institute, 2021. *Stack Canaries – Gingerly Sidestepping the Cage.* [Online]
Available at: https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/
[Accessed 17 February 2022].

Shacham, H. et al., 2007. *On the Effectiveness of Address-Space Randomization.* [Online]
Available at: https://hovav.net/ucsd/dist/asrandom.pdf
[Accessed 16 February 2022].

Su, C. & Zeng, Q., 2021. *Survey of CPU Cache-Based Side-Channel Attacks: Systematic.* [Online]
Available at: https://downloads.hindawi.com/journals/scn/2021/5559552.pdf
[Accessed 02 March 2022].

Yadav, S., Ahmad, K. & Shekhar, J., 2011. *Classification and Prevention Techniques of Buffer Overflow Attacks.* [Online]
Available at: https://www.researchgate.net/profile/Khaleel-Ahmad-3/publication/262494867_Classification_and_Prevention_Techniques_of_Buffer_Overflow_Attacks/links/02e7e537df06101f83000000/Classification-and-Prevention-Techniques-of-Buffer-Overflow-Attacks.pdf
[Accessed 22 February 2022].

Yarom, Y. & Falkner, K., 2014. *FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack.* [Online]
Available at: https://eprint.iacr.org/2013/448.pdf
[Accessed 02 March 2022].