Stack-Based Buffer Overflow Attacks

An analysis of recent stack-based buffer overflow attacks and the related security of certain systems

Peter Walker
u5539006

Contents

Section A

Question 1: stack-based buffer overflow attacks in x86 and x86_64 systems

I believe it is more difficult to achieve a buffer overflow attack against x86_64 systems (64-bit) than x86 systems (32-bit), due to the more effective security features available in 64-bit systems.

ASLR (Address Space Layout Randomisation) is much more effective in a (theoretically) 16 exabyte addressable memory (in a 64-bit system) than a 4gb addressable memory (in a 32-bit system). ASLR randomises the location of different parts of a program's memory, to reduce the likelihood of buffer overflow attacks overwriting critical data sections, accidentally or intentionally. This in turn helps to prevent RCE (remote code execution) from a threat actor, or unpredictable behaviour from a program, if they were to successfully overwrite register values or return addresses on the stack. With a much larger addressable memory, the number of locations different data sections could be stored at increases. This makes it more difficult for a threat actor to try and overwrite certain values of executable memory in x86_64 systems.

Stack canaries are also much more effective in 64-bit systems than 32-bit systems, because canaries can be longer and therefore more difficult to brute-force (randomly guess the canary every time the program is run). Canaries are randomly generated values placed on the stack between the return address and the location where variables are stored in the program. This allows the program to check for any kind of code injection where data has been overwritten, by checking the canary value before popping the return pointer from the stack. If the canary is not the same as was generated previously, then the program will terminate as the stack has been overwritten or corrupted. [1]

Shadow stacks, implemented by Intel into their most recent x86_64 systems as part of their Control Enforcement Technology, are stacks that only contain return pointers, but are not accessible from the main program. When a function is called, the return address is pushed to both the shadow and regular stack, and when returning from the function, the pointers from each stack are compared. If they are equal, then the code can resume from the pointer, but if not, then an overwrite attempt has been recognised, and the program will usually terminate. [2] [3]

The NX-bit, a feature implemented in all x86_64 systems, which allows the operating system to label certain memory locations as non-executable and only see them as data, is not present in all x86 systems.

I believe that it is more likely to be more difficult to achieve a buffer overflow attack against x86_64 systems than x86 systems due to the additional and more effective prevention methods available for 64-bit systems: 32-bit systems have less-effective versions of these security measures or may not have them at all.

Question 2: modern computer systems mitigation against buffer overflows in multiple layers

Modern computer systems have multiple safeguards to help mitigate buffer overflow attacks, whether at the compilation or execution stage.

The first mitigation possible is to ensure safe programming practices before compilation. This should be adopted by the developer, by avoiding functions such as “strcpy()” (C) and instead using “strncpy()”. This means that only data up to the size of the destination will be copied and so no overwrite of the destination can occur. Furthermore, input data should be sanitized, to escape any special characters and format strings, to avoid giving away information about memory status or disrupting execution flow.

Compilation tools such as GCC (GNU Compiler Collection) give warnings about functions that are vulnerable to attacks in the raw code when you attempt to compile it. FORTIFY_SOURCE is another tool (macro) that can be used to detect static length buffer overflow attacks during the compilation process. [4] After compilation, memory management tools such as Valgrind and AddressSanitizer can be used to detect memory problems such as reading uninitialized memory or memory being overwritten during runtime. [5] [6]

Another example is DEP (Data Execution Prevention), which is at the execution stage (this is implemented using the NX-bit). This marks certain areas of memory (e.g. the stack) as non-executable. This means if an attack manages to manipulate those areas of memory and inject malicious code, it won’t be run, and will only be interpreted as data by the program.

Indirect Branch Tracking through the “endbr64” instruction is another example of mitigation against buffer overflow attacks (implemented on Intel CPUs but has equivalents on other architectures). The CPU will raise an exception (#CP) if a set of instructions is jumped to but doesn’t start with the “endbr64” instruction. This means if a hacker managed to change the target address of an indirect jump, it is unlikely that the new code would run because it likely wouldn’t begin with that instruction, and the program would instead terminate. [7]

Finally, the RELRO (RELocation Read-Only) feature of modern systems can be implemented to make the Global Offset Table (GOT) and Procedure Linkage Table (PLT) completely read-only (if RELRO is fully implemented) or just make the GOT read-only, if partially implemented, however, format string attacks will still work. (It is not fully implemented by default as all addresses must be resolved at once and this is time consuming but is recommended for applications handling sensitive operations or data). [8]

Furthermore, there is ASLR that takes place, to randomize the positions of sections of an executable within its memory allocation, to make it harder to find the necessary information or section to overwrite.

Stack canaries are also used and placed between local variables and return addresses on the stack, to check for potential overwrites onto the stack. However, these are ineffective against attacks that selectively overwrite data on the stack. [1]

Finally, on some recent Intel CPUs, shadow stacks have been implemented. When a function is called, the return address is pushed to both the call and shadow stack, and just before returning, the two values are compared. If they are equal, the program continues to execute, and if not, the program will terminate due to an unauthorized overwrite of the stack. These are useful since programs do not have access to the shadow stack, but it is instead managed by the OS. [3]

Question 3: how side-channel attacks via CPU cache affect hardware security of a device

A side-channel attack is an attack formed from information that is collected because of the poor implementation of a protocol or algorithm and does not directly affect or interact with software or hardware vulnerabilities.

A side-channel attack via the CPU cache can involve timing attacks (studying minute changes in times taken to complete operations), which allows speculation about the data the CPU is processing. For example, by monitoring a cryptographic algorithm's operations, it may be possible to determine the secret key used. [9] There are also minute changes in cache access times due to contention which can be traced to deduce changes the program makes to the cache, such as is used for CacheBleed. [10] Another cache side-channel attack could involve flushing speculative execution cache to determine types of memory being accessed and determining the content of protected memory areas such as is used for Meltdown. [11]

A technique used in side-channel attacks via CPU cache is "flush and reload". This is implemented as part of the Meltdown exploit. The exploit starts by clearing the cache (the "flush").

Most modern devices use a CPU optimization algorithm called "speculative execution" which involves performing operations before it is known whether they will be needed. These operations are run separately from the program and are therefore not affected by its privilege levels. This means sensitive data can be accessed (even when the program itself can't access it) and may be able to affect the cache state.

After this has run for some amount of time, the cache state will be affected (the "reload"). When the exploit tries to access the shared cache, it measures the time taken to attempt to retrieve the data. By inspecting the time taken, it is possible to determine what data is from the speculative execution, then read that data using cache timing attacks. [11]

This means that it is possible for malicious programs to not only identify but also read data that is protected either from memory, or from cache. This could allow malware to obtain confidential data such as secret keys, passwords, and memory dumps.

Standard stack-based buffer overflow attacks in recent years

CVE-2024-23622

In the IBM Merge Healthcare eFilm Workstation license server, there is a stack-based buffer overflow vulnerability. According to Exodus (the company credited with the discovery) the size of the buffer was miscalculated. The allocated memory for this buffer must've been smaller than expected since it could be overflown. Any RCE has system privileges on the server. [12] [13]

CVE-2017-11882

Multiple releases and versions of the Microsoft Office Service Pack and Microsoft Office 2016 are all vulnerable to a stack-based buffer overflow, where a bad actor can execute code as the current user due to improper handling of objects in memory. This is through the Microsoft Equation Editor, due to a lack of exploit prevention such as ASLR or DEP for this standalone process. [14] [15]

CVE-2019-7232

The ABB IDAL server is susceptible to a stack-based buffer overflow attack when supplied with a long Host header, in a web request. It can overflow a buffer and overwrite a Structured Exception Handler (SEH). The Host header needs to be 2047 or more to overflow the buffer and overwrite the SEH address, which can cause execution of uploaded or unintended code. [16]

Section B

Part 1: tracing program execution and bypassing login verification during runtime

Program Execution and Trace

The program grants access using an access key to a “bug bounty” system. It has a few different access keys stored in the .data section of the assembly and loads them all into memory and then compares the program arguments with one of them to check if the user has entered the correct access key. By loading multiple potential access keys, the developer has implemented *security by obscurity*.

```
gdb-peda$ run a
Starting program: /home/kali/Binary_X a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

+-+-+-+-+ +-+ +-+-+-+ +-+-+-+-+-+-+
|C|A|S|O| |-| |B|u|g| |B|o|u|n|t|y|
+-+-+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+
Created by AP


Incorrect Key, Access Denied!
[Inferior 1 (process 15557) exited normally]
```

At the beginning, the program will exit if no arguments are provided. Assuming arguments are provided, the several potential access keys are loaded into the stack, and the user input is loaded into the RDI register.

```
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffdde0 --> 0x7fffffffdf38 --> 0x7fffffffe2b8 ("/home/kali/Binary_X")
0008| 0x7fffffffdde8 --> 0x200000000
0016| 0x7fffffffddf0 --> 0x5555555560b2 ("2112751046-43262-62922-87643")
0024| 0x7fffffffddf8 --> 0x5555555560cf ("643he6f98sh7420")
0032| 0x7fffffffde00 --> 0x5555555560df ("gf7f0gau8hehu3u")
0040| 0x7fffffffde08 --> 0x5555555560ef ("055556768429")
0048| 0x7fffffffde10 --> 0x5555555560fc ("978347120101")
0056| 0x7fffffffde18 --> 0x555555556109 ("73hrskv923j90")
[-----------------------------------------------------------------------------]
```

Afterwards, the correct access key is loaded into the RSI register and compared to RDI.

```
RSI: 0x5555555560ef ("055556768429")
RDI: 0x7fffffffe2cc --> 0x46524f4c4f430061 ('a')
```

If they have the same value, and therefore the correct access key has been entered by the user, the *strcmp()* command will return a 0. The next command checks the EAX register (RAX but the 32-bit version) which stores the output of the *strcmp()* command, and if the return value was 0, then it sets the zero flag to 1 (it is a 0).

```
gdb-peda$ info registers eflags
eflags         0x246                [ PF ZF IF ]
gdb-peda$
```

If the access keys are not the same, the same process runs but EAX will not contain a 0. This means a 0 will be set as the zero flag as the value was not equal to 0.

The program then checks the zero flag, and if it equal to 0, grants the user access by not jumping. If not 0, then the program will jump to a later instruction and deny the user access and display an appropriate message.

Using GDB

The first thing to do when analyzing any binary file is disassembling the “main” function. This gives us some idea of functions called and what the program does.

```
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>
   0x0000555555555218 <+143>:   lea    rax,[rip+0xe93]        # 0x5555555560b2
   0x000055555555521f <+150>:   mov    QWORD PTR [rbp-0x30],rax
   0x0000555555555223 <+154>:   lea    rax,[rip+0xea5]        # 0x5555555560cf
   0x000055555555522a <+161>:   mov    QWORD PTR [rbp-0x28],rax
   0x000055555555522e <+165>:   lea    rax,[rip+0xeaa]        # 0x5555555560df
   0x0000555555555235 <+172>:   mov    QWORD PTR [rbp-0x20],rax
   0x0000555555555239 <+176>:   lea    rax,[rip+0xeaf]        # 0x5555555560ef
   0x0000555555555240 <+183>:   mov    QWORD PTR [rbp-0x18],rax
   0x0000555555555244 <+187>:   lea    rax,[rip+0xeb1]        # 0x5555555560fc
   0x000055555555524b <+194>:   mov    QWORD PTR [rbp-0x10],rax
   0x000055555555524f <+198>:   lea    rax,[rip+0xeb3]        # 0x555555556109
   0x0000555555555256 <+205>:   mov    QWORD PTR [rbp-0x8],rax
   0x000055555555525a <+209>:   mov    rax,QWORD PTR [rbp-0x40]
   0x000055555555525e <+213>:   add    rax,0x8
   0x0000555555555262 <+217>:   mov    rax,QWORD PTR [rax]
   0x0000555555555265 <+220>:   mov    rdx,QWORD PTR [rbp-0x18]
   0x0000555555555269 <+224>:   mov    rsi,rdx
   0x000055555555526c <+227>:   mov    rdi,rax
   0x000055555555526f <+230>:   call   0x555555555080 <strcmp@plt>
   0x0000555555555274 <+235>:   test   eax,eax
   0x0000555555555276 <+237>:   jne    0x555555555292 <main+265>
   0x0000555555555278 <+239>:   lea    rdi,[rip+0xe98]        # 0x555555556117
   0x000055555555527f <+246>:   call   0x555555555070 <puts@plt>
   0x0000555555555284 <+251>:   lea    rdi,[rip+0xe9c]        # 0x555555556127
   0x000055555555528b <+258>:   call   0x555555555070 <puts@plt>
   0x0000555555555290 <+263>:   jmp    0x55555555529e <main+277>
   0x0000555555555292 <+265>:   lea    rdi,[rip+0xeaf]        # 0x555555556148
   0x0000555555555299 <+272>:   call   0x555555555070 <puts@plt>
   0x000055555555529e <+277>:   mov    eax,0x0
   0x00005555555552a3 <+282>:   leave
   0x00005555555552a4 <+283>:   ret
End of assembler dump.
```

In this program, the function labelled as “<puts@plt>” is run many times, and this is used to display data to the console. This function is called from the PLT (Procedure Linkage Table) which contains default functions references, from dynamically linked libraries used in runtime.

```
0x000055555555527f <+246>:    call    0x555555555070 <puts@plt>
```

We can also see the “<strcmp@plt>” command included which compares two strings (in this case the access key and the user input). This is used as a condition for the next statement “jne 0x1292” which may cause the program to jump to a different instruction depending on the result of the comparison.

```
0x000055555555526f <+230>:    call    0x555555555080 <strcmp@plt>
```

When we run the program with user input, and a breakpoint before *strcmp(),* we can see both the access key and the user input stored in RSI and RDI registers, respectively.

```
RSI: 0x5555555560ef ("055556768429")
RDI: 0x7fffffffe2cc ⟶ 0x46524f4c4f430061 ('a')
```

Now that we can see the two values that are going to be compared, since they are loaded into registers, since the *strcmp()* command compares the RSI and RDI registers.

This gives us the correct access key which can be entered into the program to grant immediate access.

Using CLI

It is also possible to use the *strings* command in the linux terminal to search for strings in the binary file which outputs the following:

```
┌──(kali㉿kali)-[~]
└─$ strings Binary_X
/lib64/ld-linux-x86-64.so.2
libc.so.6
exit
puts
__cxa_finalize
strcmp
__libc_start_main
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u+UH
[]A\A]A^A_
+-+-+-+-+ +-+ +-+-+-+ +-+-+-+-+-+-+
|C|A|S|O| |-| |B|u|g| |B|o|u|n|t|y|
+-+-+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+
Created by AP
How to run me : ./Binary_X <KEY>
2112751046-43262-62922-87643
643he6f98sh7420
gf7f0gau8hehu3u
055556768429
978347120101
73hrskv923j90
Access Granted
Hint: Please secure me !!!
Incorrect Key, Access Denied!
```

There are 6 arbitrary strings included in the output, which allows any unauthorized user to brute force the login with these 6 strings until they guess the correct one.

Bypassing the login using a hexdump

Currently, the program jumps to an access denied clause if the two strings don't match (and therefore the zero flag is 0). If the "jne" opcode was changed to "je" then the program would only jump to the access denied statement if the two strings DID match.

We first need to find the address of the jne command within the program. This is achieved by disassembling the "main" function inside gdb, before running the program. This means that the address space has not yet been randomized and so the address will be relative to the start of the program.

If we check the assembly docs for the hex of "jne" and "je" we find out that they are "0x75" and "0x74" respectively.

If we then hex dump the binary file, we can edit the hex byte at 0x1276 to 0x74 from 0x75.

```
┌──(kali㉿kali)-[~]
└─$ xxd Binary_X > Bin_Hexdump

┌──(kali㉿kali)-[~]
└─$ nano Bin_Hexdump
```

```
000011f0: 7cfe ffff 488d 3d35 0e00 00e8 70fe ffff  |...H.=5....p...
00001200: 488d 3d51 0e00 00e8 64fe ffff 488d 3d69  H.=Q....d...H.=i
00001210: 0e00 00e8 58fe ffff 488d 0593 0e00 0048  ....X...H......H
00001220: 8945 d048 8d05 a50e 0000 4889 45d8 488d  .E.H......H.E.H.
00001230: 05aa 0e00 0048 8945 e048 8d05 af0e 0000  .....H.E.H......
00001240: 4889 45e8 488d 05b1 0e00 0048 8945 f048  H.E.H......H.E.H
00001250: 8d05 b30e 0000 4889 45f8 488b 45c0 4883  ......H.E.H.E.H.
00001260: c008 488b 0048 8b55 e848 89d6 4889 c7e8  ..H..H.U.H..H...
00001270: 0cfe ffff 85c0 741a 488d 3d98 0e00 00e8  ......u.H.=.....
00001280: ecfd ffff 488d 3d9c 0e00 00e8 e0fd ffff  ....H.=.........
00001290: eb0c 488d 3daf 0e00 00e8 d2fd ffff b800  ..H.=...........
000012a0: 0000 00c9 c366 2e0f 1f84 0000 0000 0090  .....f..........
000012b0: f30f 1efa 4157 4c8d 3deb 2a00 0041 5649  ....AWL.=.*..AVI
000012c0: 89d6 4155 4989 f541 5441 89fc 5548 8d2d  ..AUI..ATA..UH.-
000012d0: dc2a 0000 534c 29fd 4883 ec08 e81f fdff  .*..SL).H.......
000012e0: ff48 c1fd 0374 1f31 db0f 1f80 0000 0000  .H...t.1........
000012f0: 4c89 f24c 89ee 4489 e741 ff14 df48 83c3  L..L..D..A...H..
00001300: 0148 39dd 75ea 4883 c408 5b5d 415c 415d  .H9.u.H...[]A\A]
00001310: 415e 415f c366 662e 0f1f 8400 0000 0000  A^A_.ff.........
00001320: f30f 1efa c300 0000 f30f 1efa 4883 ec08  ............H...
```

If we then recompile and run the program, we can enter any value (as long as it doesn't match the actual access key!) and we will be granted access.

```
┌──(kali㉿kali)-[~]
└─$ xxd -r Bin_Hexdump > Binary_X_Cracked
```

```
┌──(kali㉿kali)-[~]
└─$ ./Binary_X_Cracked aaaaa

+-+-+-+-+ +-+ +-+-+-+ +-+-+-+-+-+-+
|C|A|S|O| |-| |B|u|g| |B|o|u|n|t|y|
+-+-+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+
Created by AP


Access Granted

Hint: Please secure me !!!
```

Part 2: alternative solution to proposed weakness in the binary file

The problem with this binary file is that the access key (password) is stored in plaintext in the binary file.

Hashing the user input and comparing to existing hashes in the binary file would be a much more secure way of checking the access key, as hashing is one-way encryption, so reading the hash would be useless to anyone trying to exploit the file as they couldn't reverse it to find the plaintext.

A further security measure would be to implement salting for the hashes so that rainbow-table attacks are pointless. This involves adding a random string of some characters to the end of the access key before hashing then storing that along with the hash and appending to the user input to check if the hashes match.

Finally, it may be worth adding some time delay to the program or using a slow hashing algorithm to reduce the attempt rate, such as bcrypt. This would help mitigate brute force attacks against the file.

To prevent editing the binary file itself (as we proposed), the program would need to be signed by the developer/distributor to verify integrity. This would mean if it was edited, the signature would no longer match, and this would flag an error on the user's system when downloading etc. However, this would not stop users modifying their local files, so it important to isolate the binary from a global server, but have the program linked to a user's local database.

References

[1] M. Lemmens, “Stack Canaries - Gingerly Sidestepping the Cage,” SANS, 04 02 2021. [Online]. Available: https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/. [Accessed 06 02 2024].

[2] “A Technical Look at Intel’s Control-flow Enforcement Technology,” Intel, 13 06 2020. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html. [Accessed 06 02 2024].

[3] “Control-flow Enforcement Technology (CET) Shadow Stack,” The Linux Kernel, 28 04 2023. [Online]. Available: https://www.kernel.org/doc/html/next/x86/shstk.html. [Accessed 06 02 2024].

[4] “Enhance application security with FORTIFY_SOURCE,” Red Hat, 26 03 2014. [Online]. Available: https://www.redhat.com/en/blog/enhance-application-security-fortifysource. [Accessed 06 02 2024].

[5] “About Valgrind,” Valgrind Developers, [Online]. Available: https://valgrind.org/info/. [Accessed 06 02 2024].

[6] “AddressSanitizer,” Microsoft, 14 06 2023. [Online]. Available: https://learn.microsoft.com/en-us/cpp/sanitizers/asan?view=msvc-170. [Accessed 06 02 2024].

[7] Intel, “Indirect Branch Tracking,” in *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, 2023, pp. 411-418.

[8] “Hardening ELF binaries using Relocation Read-Only (RELRO),” Red Hat, 28 01 2019. [Online]. Available: https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro. [Accessed 06 02 2024].

[9] D. Brumley and D. Boneh, “Remote Timing Attacks are Practical,” 2003. [Online]. Available: https://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf. [Accessed 06 02 2024].

[10] Y. Yarom, D. Genkin and N. Heninger, “CacheBleed: A Timing Attack on OpenSSL Constant,” [Online]. Available: https://eprint.iacr.org/2016/224.pdf. [Accessed 06 02 2024].

[11] M. Lipp et al., “Meltdown: Reading Kernel Memory from User Space,” [Online]. Available: https://meltdownattack.com/meltdown.pdf. [Accessed 06 02 2024].

[12] “CVE-2024-23622 Detail,” NIST, 25 01 2024. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-23622#range-10268449. [Accessed 06 02 2024].

[13] “CWE-131: Incorrect Calculation of Buffer Size,” Common Weakness Enumeration, 19 07 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/131.html. [Accessed 06 02 2024].

[14] “CVE-2017-11882 Detail,” NIST, 14 11 2017. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2017-11882. [Accessed 06 02 2024].

[15] “CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer,” Common Weakness Enumeration, 19 07 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/119.html. [Accessed 06 02 2024].

[16] “CVE-2019-7232 Detail,” NIST, 24 06 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-7232. [Accessed 06 02 2024].