CH 10

Monday, January 7, 2019 2:22 PM

Fuzzing: Feeds random/not-so-random data into protocol to force processing app to crash in order to ID vulns

- Yields results no matter complexity
- Produces simple multiple test cases: Sent to app for processing
- Can be generated auto using random mods/under direction from analyst

Simplest: Sends random garbage to see what happens: **cat /dev/urandom | nc hostname port**

- Reads data from system's RNG device using cat: Piped into netcat: Opens connection as instructed

Mutation Fuzzer: Using existing protocol data/mutate it in some way/send it to receiving app

Simplest: Random bit flipper

```
void SimpleFuzzer(cons char* data, size_t length) {
    size_t position = RandomInt(length);
    size_t bit = RandomInt(8);

    char* copy = CopyData(data, length);
    copy[position] ^= (1 << bit);
    SendData(copy, length);
}
```

1. SimpleFuzzer() function: Takes in data/length of data to fuzz:
   - Generates random num bet 0/length of data to mod
2. Decides which bit in byte to change by generating num between 0-7
   - Toggles bit using XOR/sends mutated data to network destination

Vulnerability Triaging: Taking a series of steps to search for root cause of a crash

Debugging Applications: Diff platforms allow diff lvls of control over triaging: Can attach debugger to process

Cmds: Running debuggers on Win/Linux/MacOS

Debugger	New Process	Attach process
CDB (Win)	cdb application.exe [args]	cdb -p PID
GDB (Linux)	gdb --args application [args]	gdb -p PID
LLDB (macOS)	lldb -- application [args]	lldb -p PID

- Debugger will suspend execution of process after you've created/attached debugger: Run process again

Simplified App Execution Cmds

Debugger	Start Execution	Resume Execution
CDB	g	g
GDB	run, r	continue, c
LLDB	process launch, run, r	thread continue, c

When new process creates child process: Might be child process that crashes instead of one debugging

- Can follow child/not parent

Debugging Child Processes

Debugger	Enabled child process debugging	Disable child process debugging
CDB	.childdbg 1	.childdbg 0
GDB	set follow-fork-mode child	set follow-fork-mode parent
LLDB	process attach --name NAME --waitfor	exit debugger

Analyzing the Crash: Look for crashes that indicate corrupted mem:

Windows: Access violation | Linux: SIGSEGV

Instruction Disassembly Commands

Debugger	Disassemble from crash location	Disassemble from specific location
CDB	u	u ADDR

GDB	disassemble	disassemble ADDR
LLDB	disassemble -frame	disassemble --start-address ADDR

Displaying/Setting Processor Register State

Debugger	Show general purpose registers	Show specific registers	Set specific register
CDB	r	r @rcx	r @rcx = NEWVALUE
GDB	info registers	info registers rcx	set $rcx = NEWVALUE
LLDB	register read	register read rcx	register write rcx NEWVALUE

- Can use these to set the value of register: Allows you to keep app running by fixing crash/restarting execution

Creating a Stack Trace: When app debugging crashes: Want to display how current function was called

- Can narrow down which parts of protocol needed to focus on reproducing crash
- Can get context by generating stack trace
- Displayed functions called prior to execution of vuln: Including some local vars/args passed to them

Creating a Stack Trace

Debugger	Display stack trace	Display stack trace with arguments
CDB	K	Kb
GDB	backtrace	backtrace full
LLDB	backtrace	

Displaying Memory Values

Debugger	Display bytes/words/dwords/qwords	Display ten 1-byte values
CDB	db, dw, dd, dq ADDR	db ADDR L10
GDB	x/b, x/h, x/w, x/g ADDR	x/10b ADDR
LLDB	memory read --size 1,2,4,8	memory read --size 1 --count 10

CMDS for Displaying Process Mem Map

Debugger	Display process memory map
CDB	!address
GDB	info proc mappings
LLDB	No direct equivalent

- Determines what type of mem an addr corresponds to: Heap/stack/mapped executable
- Helps narrow down type of issue
- Example: Memory value corruption occurred? Distinguish whether stack/heap mem corruption

Rebuilding apps w/Addr Sanitizer:

Asan	Address Sanitizer: Extension for CLANG C compiler: Detects mem corruption bugs **-fsantize=address when running compiler:** ▪ Specify option using CFLAGS env var ▪ Rebuilt app will have addl instrumentation to detect common mem errors ▪ Mem corruption/out-of-bound writes/use-after-free/double-free ▪ Stops app as soon as vuln condition has occurred
Page Heap	Win Access to source code of app more restricted: ▪ **Page Heap:** Can enable chances of tracking down mem corruption **gflags.exe -i appname.exe +hpa** ▪ Comes installed w/CDB debugger ▪ **-i**: specify img filename to enable page heap on ▪ **+hpa**: What actually enables page heap when app executes **Works by allocating special OS-defined mem pages:** AKA: **guard pages** after every heap allocation ▪ If an app tries to read/write these guard pages: Error will be raised/debugger notified ▪ Useful for detecting heap overflows ▪ If overflow writes immediately at end of buffer: Guard page will be touched by app/error **Cons:** Wastes a huge amt of mem b/c each allocation needs a separate guard page

- Requires a syscall which reduces allocation performance

Exploiting Common Vulns

Stack Overflows	Occurs when code underestimates length of buffer to cp into a loc on the stack ▪ Many archs: Return addr for function stored on stack/corruption of ret addr gives direct execution ▪ Corrupt ret addr on stack to point to buffer containing shell code w/instructions ▪ Need to craft data into overflowed buffer to ensure rt addr points to mem region you control ▪ If caused by C-style str copy: Won't be able to use multiple 0 bytes in overflow ▪ **C uses a 0 byte as terminating char for string** ▪ Overflow will stop immediately ▪ Direct shell code to addr value with no 0 bytes
Heap Overflows	Often less predictable mem addr: No guarantee ▪ Exploit the structure of C++ objects: specifically Vtables **VTable:** List of pointers to functions that the object implements ▪ Allows dev to make new classes derived from existing base classes/override some functionality ▪ Each allocated instance of a class must contain a ptr to the mem loc of the function table ▪ When virtual func called on object: Compiler generates code that looks up addr of Vtable ▪ Then looks up virtual function inside table/calls addr ▪ Can't corrupt the ptrs in the table: Likely stored in read-only part of mem ▪ CAN corrupt ptr to the Vtable to gain code execution
Use-After-Free	Corruption of the state of the program/not exactly mem ▪ When mem block is freed but ptr to block stored by some part of app ▪ Later in app execution: ptr to freed block re-used <u>Bet time mem block freed/ptr reused opportunity to replace contents of block w/arbitrary values</u> ▪ Gain code execution ▪ When mem block freed: Will be given back to heap to be reused for another mem allocation ▪ As long as you can issue allocation req of same size as original allocation ○ Strong possibility freed mem block would be reused w/your crafted contents **<u>App first allocations an object p on heap:</u>** Contains a Vtable ptr we want to control ▪ App calls del on ptr to free mem ▪ App doesn't reset value of p: Object free to be reused in the future ▪ Exploit allocates mem of exact size/has control over contents of mem p points to ▪ Heap allocator reuses as allocation for p ▪ If app reuses p to call a virtual function: Can control lookup/gain execution

Manipulating Heap Layout: Key to success usually is in forcing suitable allocation to occur at a reliable loc
- Heap implementation for an app may be based on virtual mem mgmt features of platform app exe on

<u>Using OS virtual mem allocator has problems:</u>
- Poor perf: Each allocation/free-up requires OS to switch to kernel mode/back
- Wasted mem: Virtual mem allocations done at page level: At least 4096 bytes
  - If you allocate mem smaller than page size: Rest of page wasted

Free-list	Maintains a list of freed allocations inside a larger allocation ▪ When allocation req made: ▪ Heap's implementation scans list of free blocks looking for sufficient size ▪ Would use free block/allocate req block at start ▪ Update free-list to reflect new free size
Defined Mem Pools	Defined mem pools for diff allocations sizes: • Groups smaller allocations appropriately • When req made: Implementation will allocate buffer based on pool most closely matched • Reduces fragmentation caused by small allocations
Heap mem storage	How info like free-list stored in mem: 2 methods

1. **In-band:** Metadata (block size): Whether state is free/allocated stored along allocated mem
2. **Out-of-band:** Metadata stored elsewhere in mem: Easier to exploit
   - Don't have to worry about restoring impt metadata when corrupting mem blocks
   - Useful when you don't know what values to restore for metadata to be valid

Arbitrary Mem Write Vuln: File write resulting from incorrect resource handling
- May be due to cmd that allows you to specify loc of a file write/path canonicalization
- Could occur as a by-product of another vuln like heap overflow
- Many old heap mem allocators would use linked list structure to store list of free blocks
- If linked list corrupted: Any mod of free-list could result in arbitrary write of value into attacker-supplied loc

To exploit: Need to mod loc that can directly control code execution
- Could target Vtable ptr of an object in mem/overwrite to gain control over execution

Advantage: Can lead to subverting logic of an app

Mitigating Mem Corruption:

DEP/NX	**Data Execution Prevention/No-Execute:** ▪ Attempts to mitigate by req mem w/executable instructions to be specifically allocated by OS ▪ Requires processor support so if process tries to execute mem at addr not marked: Raises error ▪ OS terminates process in error to prevent further execution Can determine whether executable mem is being used through memory mapping cmds ▪ If DEP enabled: Can use ROP: Return-Oriented Programming as a workaround
ROP	**Return-Oriented Programming** ▪ Repurposes existing already executable restructures rather than injecting arbitrary instructions ▪ Sequence of instructions doesn't have to execute as originally compiled into code ▪ Can make small snippets of code throughout program **ROP gadgets:** These small sequences of instructions ▪ Easier when you have a stack overflow ▪ Heap overflow? Will need a stack pivot **Stack pivot:** ROP gadget that allows you to set current stack ptr to known value
ASLR	**Address Space Layout Randomization:** ▪ Bypassing DEP became more diff: Randomizes the layout of a processes addr space ▪ Makes it harder to predict ▪ Location of an exe in ASLR isn't always randomized bet 2 separate processes ▫ Vuln that could disclose loc of mem **Partial overwrites:** Lower bits of random mem ptrs can be predictable if upper bits totally random
Canaries	Detect corruption/immediately cause app to terminate: ▪ Random number generated by app during startup: Stored in global mem loc ▪ Can be accessed by all code in app ▪ Random num pushed onto stack when entering a function ▪ When function exist: Random value popped off stack/compared to global value ▪ If global value doesn't match what was popped: App assumes stack mem corrupted/terminates **Bypassing:** Typically only protect the ret addr of currently executing func on stack ▪ If stack overflow has controlled length: Possible to overwrite these vars w/out corrupting canary ▪ Buffer underflow