CSE 127: Computer Security

Memory Integrity

Kirill Levchenko

November 18, 2014
Stack Buffer Overflow

- **Stack buffer overflow**: writing past end of a stack-allocated buffer
  - Also called *stack smashing*

- One of the simplest control flow hijacking mechanisms

- Stack buffer overflow vulnerabilities still exist
  - Mainly embedded devices and legacy software
  - Newer OSes provide protection against common types
NOVEMBER 4, 2014

CVE-2014-1635 BELKIN N750 BUFFER OVERFLOW

1. Vulnerability Properties

Title: Belkin n750 buffer overflow

CVE ID: CVE-2014-1635

CVSSv2 Base Score: 10 (AV:N/AC:L/AU:N/C:P/I:N/A:N)

Vendor: BELKIN (http://www.belkin.com/)

Products: n750/F9K1103

Advisory Release Date: 2014-11-04

Advisory URL: https://labs.integrity.pt/advisories/CVE-2014-1635/

Credits: Discovery and PoC by Marco Vaz <mv[at]integrity.pt>

2. Vulnerability Summary

A remote unauthenticated attacker may execute commands as root by sending an unauthenticated crafted POST request to the HTTP that uses no authentication on the protocol network.
Buffer Overflow Defenses

- Avoid unsafe functions
- Stack canary
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both ($W^X$)
- Control flow integrity (CFI)
Avoiding Unsafe Functions

- `strcpy`, `strcat`, `gets`, *etc.*

- **Plus:** Good idea in general

- **Minus:** Requires manual code rewrite

- **Minus:** Non-library functions may be vulnerable also
  - E.g. user creates her own `strcpy`

- **Minus:** No guarantee you found everything
Stack Canary

- Special value placed before return address on stack
  - Secret random value chosen at program start
  - String terminator ‘\0’

- Gets overwritten during buffer overflow

- Check canary before jumping to return address

- Automatically inserted by compiler
  - **GCC:** `-fstack-protector` or `-fstack-protector-strong`
Stack Canary

Stack before overflow with canary
- Stack grows from higher to lower addresses
- char *buffer[20] CY RA parameters other data
- Buffer writes from lower to higher addresses
- Low address to high address
- RA = return address
- CY = canary

Stack after overflow attack with destroyed canary
- Stack grows from higher to lower addresses
- NOP sled shell code RA RA RA RA RA RA RA other data
- Destroyed canary indicating stack smashing
- Payload (cuckoo’s egg)
- Buffer has grown over its boundaries
- Low address to high address

(source: Dr. Dobb’s)
Stack Canary

- **Plus:** No code changes required, only recompile
- **Minus:** Performance penalty before every return
- **Minus:** Only protects against stack buffer overflows
- **Minus:** Random canary fails if attacker can read memory
ASLR
Address Space Layout Randomization

- Change location of stack, heap, code, static variables
- Works because attacker needs address of shellcode
- Layout must be unknown to attacker
  - Randomize on every launch (best)
  - Randomize at compile time
- 32-bit address space and alignment limits entropy
  - E.g. 4KB page-aligned stack leaves only 20 bits to randomize
- Implemented on most modern OSes in some form
ASLR
Address Space Layout Randomization

- **Plus:** No code changes or recompile required
- **Minus:** 32-bit architectures get limited protection
- **Minus:** Fails if attacker can read memory
- **Minus:** Load-time overhead for strongest protection
  - No executable image sharing between processes
Use hardware memory protection to ensure memory cannot be both writeable and executable at the same time.

- Code is executable, not writeable.
- Stack, heap, static variables are writeable, not executable.
- Supported by most modern processors:
  - AMD64 and later; also available via segments since 80286.
- Implemented by modern operating systems.
W^X

- **Plus:** No code changes or recompile required
- **Minus:** Requires hardware support
- **Minus:** Breaks self-modifying code (extremely rare)
- **Minus:** Defeated by return-oriented programming
- **Minus:** Does not protect just-in-time compiled code
Return to Libc

- **Problem**: If stack not executable, can’t use shellcode
- **Solution**: Jump into existing program code!
- Search executable for code that does what you want
  - E.g. if executable calls `exec("/bin/sh")` you’re done
- Need known executable – usually not a problem
- Hindered by ASLR
Return-Oriented Programming

- **Problem:** Executable may not have single code fragment that does exactly what you want
- **Solution:** String *multiple* pieces of existing code
ROP “gadgets”

- Fragments of code that return control back to attacker
- Instruction sequences that end in \texttt{ret} op
  - Found at end of function or as unintended instruction (CISC)
- The \texttt{ret} instruction pops address off stack, jumps to it
objdump -d /bin/sh

150af: 48 83 c0 01  
   add $0x1,%rax

150b3: c3  
   retq

150ba: 48 83 c4 08  
   add $0x8,%rsp

150be: c3  
   retq

1461b: 31 c0  
   xor %eax,%eax

1461d: 5b  
   pop %rbx

1461e: c3  
   retq
Return Oriented Program

- Overwrite stack with sequence of gadget addresses
- Each ret advances stack pointer to next address
- In effect, stack becomes program, SP becomes IP

```
c
b
a
ret
sfp
```

```
ret4
ret3
ret2
ret1
sfp
```

← ESP
What if we can’t insert shellcode on stack?
Heap Spray

- What can we do if we can only modify IP?
- ROP requires ropcode to be placed on stack
- Place instructions into program heap!
- Give target shellcode as input
  - Most programs will store user input in a heap variable
- If memory allocation highly deterministic, location of shellcode can be guessed
JIT Spray

- But what if target uses $W^X$?
- Just-in-time compilers produce data that becomes executable code
- JIT spray: provide target a program that is compiled into exploit able native code