Binary Code Analysis and Secure Software Systems

07 – Canary & Memory Disclosure

Sang Kil Cha
Control Hijack Attack / Defense

- Direct code injection
- NX / DEP
- Code-reuse attacks (e.g., ROP)
- ASLR
- Exploiting fixed code section with ROP

- Canary
- Fine-grained ASLR
Control Flow Hijack Attack

Corrupted Memory

Attacker’s Code (Shellcode)

Hijacked Control Flow

DEP makes this region non-executable!
Different Perspective: ASLR

Corrupted Memory

Hijacked Control Flow

Attacker’s Code (Shellcode)

ASLR makes it difficult to guess the address of the shellcode
Another Perspective: Canary

Attacker’s Code (Shellcode) → Corrupted Memory

Hijacked Control Flow

Canary makes it difficult to hijack the control flow
Canary
Canary in a Cole Mine
Mitigating Buffer Overflows with Canary

• First introduced in 1998.


• Similar approach called *StackShield* was introduced in 1999.

• Not necessarily used for stack, but can also be used for heap
Stack Canary (a.k.a. Stack Cookie)

Without Stack Canary

With Stack Canary

Check before executing return!
StackGuard (1998)

Uses a constant canary value 0x000aff0d

- Stops `strcpy`
- Stops EOF checks
- Stops `fgets`
Problem of Constant Canary Value

`memcpy`?
Random Canaries

Pick a random value at process initialization.
Problems Still Exist

Local variables are not protected!

...  
Return addr.  
Old EBP  
Canary Value  
Function Ptr.  
buf
Reordering Local Variables

Always put local buffers after local pointers.

(This idea is implemented in GCC 4.1 in 2005.)
Without Stack Canary

```assembly
80483fb: push ebp
80483fc: mov ebp, esp
80483fe: sub esp, 0x100
8048404: push DWORD PTR [ebp+0x8]
8048407: lea eax, [ebp-0x100]
804840d: push eax
804840e: call 80482d0 <strcpy@plt>
8048413: add esp, 0x8
8048416: leave
8048417: ret
```

With Stack Canary

```assembly
804844b: push ebp
804844c: mov ebp, esp
804844e: sub esp, 0x108
8048454: mov eax, DWORD PTR [ebp+0x8]
8048457: mov DWORD PTR [ebp-0x108], eax
804845d: mov eax, gs:0x14
8048463: mov DWORD PTR [ebp-0x4], eax
8048466: xor eax, eax
8048468: push DWORD PTR [ebp-0x108]
804846e: lea eax, [ebp-0x104]
8048474: push eax
8048475: call 8048320 <strcpy@plt>
804847a: add esp, 0x8
804847d: mov eax, DWORD PTR [ebp-0x4]
8048480: xor eax, DWORD PTR gs:0x14
8048487: je 804848e <somefn+0x43>
8048489: call 8048310 <__stack_chk_fail@plt>
804848e: leave
804848f: ret
```

 gcc -fno-stack-protector

 gcc -fstack-protector
Random canary value at gs:0x14

Why?

With Stack Canary

gcc -fstack-protector
GS Segment Register?

- x86 maintains a Local Descriptor Table (LDT) in memory.
- Segment registers hold an offset of the LDT.
- On Linux, GS segment register points to an entry of LDT, which represents Thread Control Block (TCB).

```c
typedef struct {
    void *tcb;      /* gs:0x00 Pointer to the TCB. */
    dtv_t *dtv;     /* gs:0x04 */
    void *self;     /* gs:0x08 Pointer to the thread descriptor. */
    int multiple_threads; /* gs:0x0c */
    uintptr_t sysinfo; /* gs:0x10 Syscall interface */
    uintptr_t stack_guard; /* gs:0x14 Random value used for stack protection */
    uintptr_t pointer_guard; /* gs:0x18 Random value used for pointer protection */
    int gscope_flag; /* gs:0x1c */
    int private_futex; /* gs:0x20 */
    void *__private_tm[4]; /* gs:0x24 Reservation of some values for the TM ABI. */
    void *__private_ss; /* gs:0x34 GCC split stack support. */
} tcbhead_t;
```
Who Initializes \([\text{gs:0x14}]\)?

Runtime Dynamic Linker (RTLD) does it every time it launches a process

```c
// Below is roughly what RTLD does at process creation time
uintptr_t ret;
int fd = open("/dev/urandom", O_RDONLY);
if (fd >= 0) {
    ssize_t len = read(fd, &ret, sizeof(ret));
    if (len == (ssize_t)sizeof(ret)) {
        // inlined assembly for moving \text{ret} to \([\text{gs:0x14}]\)
    }
}
```
GCC ProPolice Implementation

• Uses random canary values for every process creation
• Puts buffers after any local pointers on the stack
Attacking Canary Protection
Reused Canary Value

- accept() → fork() → read()
  - parent
  - child
  - Canary value is the same for every child

vs.

- accept() → fork() → execve()
  - parent
  - child
  - Canary value changed

e.g., OpenSSH does this
Attack #1: Byte-by-Byte Brute Forcing

- accept()
- fork()
- read()

Random Canary

0x429af70c
Old EBP
Return addr.
...
Attack #1: Byte-by-Byte Brute Forcing

Try to overwrite only 1 byte with a character from $\text{x00}$ to $\text{xff}$ until the program does not crash.
Attack #1: Byte-by-Byte Brute Forcing

Do the same for all 4 bytes! Worst case: 256 x 4 iterations
Any Problem?

Brute-forcing may not work if

(1) canary contains a character that we cannot use
   (e.g., a NULL byte in canary for strcpy overflows)
(2) we cannot control the last byte of the buffer
char *bp = buf;
while (buflen) {
    toread = pr_netio_read(in_nstrm, pbuf->buf,
        (buflen < pbuf->buflen ? buflen : pbuf->buflen), 1);
    while (buflen && toread > 0 && *pbuf->current != '\n' && toread--)
        {
            ...
        }
    if ( *bp == TELNET_IAC ) {
        /* \x0 = 0xFF */
    ...
    buflen--;
    telnet_mode = 0;
    break;
}
...
bp += 1;
buflen--;
}
*bp = '\0';
return buf;

ProFTPd Example (CVE-2010-3867)

1. buflen = 1
2. Buffer starts with 0xff
3. buflen becomes 0
4. buflen becomes -1
5. Integer overflow allows overwriting local buf

Problem: we cannot control the last byte!
Protecting Canary Brute-Forcing Attack

(Optional Reading)
DynaGuard: Armoring Canary-based Protections against Brute-force Attacks,
ACSAC 2015
Canary

- First canary defense (StackGuard)
- StackGuard and StackShield Attacks
- Re-implementing random canary (GCC integration)
- Byte-by-byte attack (Brute-forcing)
- ...
- Brute-forcing Protection (ACSAC 2015)

Other Attack Possibilities

• If there is another vulnerability that allows us to leak stack contents, we can easily bypass canary checks

• Canary is inherently vulnerable to format string attacks
Memory Disclosure
Memory Corruption vs. Disclosure

Memory disclosure does not necessarily involve memory corruption.
Overflow vs. Over-Read

Buffer over-read is a bug that allows an attacker to read beyond the size of a buffer.

Over-read does not corrupt memory!
Example: Heartbleed Bug (in 2014)

• OpenSSL
  − TLS *heartbeat* implementation bug

• An attacker can steal private keys
Example: Heartbleed Bug (in 2014)

Are you still there? If so reply a 5-byte string “IS561”

Client

IS561

Server
Example: Heartbleed Bug (in 2014)

Client

---

Are you still there? If so reply a **5000-byte** string “IS561”

Server

---

IS561xxxxxx...
The Bug

struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    opaque payload[HeartbeatMessage.payload_length];
    opaque padding[padding_length];
} HeartbeatMessage;

struct {
    unsigned int length;
    unsigned char *data;
    ...
} SSL3_RECORD;

Points to a message

Copy arbitrary memory contents of a server! TLS secret key may be available.

length = // obtained from // SSL3_RECORD
pl = // payload obtained // from HeartbeatMessage
memcpy(bp, pl, length);
Other Memory Disclosure

• Format string vulnerability also leaks memory info
  − “%08x.%08x.%08x…”

• Memory corruption bugs may allow memory leak
  − E.g., overwriting the length field of a string object
Memory Disclosure and Exploit

• We can bypass canary protection with memory leak.

• We can also **bypass ASLR** with memory leak.
  – Often times, control flow hijack exploitation requires two vulnerabilities: one for leaking information, and another for hijacking the control.
History So Far ...

- Direct code injection
- NX / DEP
- Code-reuse attacks (e.g., ROP)
- ASLR
- Exploiting fixed code section with ROP
- Canary
- w/ Memory Leak
- Fine-grained ASLR
JIT ROP
Just-In-Time ROP (JIT ROP)

Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization,

*IEEE S&P 2013*
JIT ROP Overview

- Use a memory disclosure bug to get the process image
  - Assumption: there is a **leaked function pointer** (memory disclosure) that we can use to read arbitrary memory addresses.

- Find ROP gadgets

- Compile ROP program for exploitation
How to Obtain the Process Image without Crashing the Program?

Leaked function pointer (ptr)

... 
cmp eax, 0x42
je 0xf00d
mov ebx, [ecx]
...

Initial code page

... 
cmp eax, 0x42
je 0xf00d
mov ebx, [ecx]
...
JIT ROP May Not Work with Oxymoron (USENIX Security 2014)

Leaked function pointer (ptr)

... cmp eax, 0x42
    je fs:[0x10]
    mov ebx, [ecx]
...

Initial code page

Difficult to retrieve jump table

But, JIT ROP is still possible when we can leak other types of pointers, e.g., vtable ptrs
Traditional Exploit Development

1. Analyze binary offline
2. Develop control hijack exploitation
3. Exploit

Target Binary

Attacker

Victim

Got a Shell

\x31\xc0\x50\x68...
JIT ROP Exploitation

1. Analyze binary offline
2. Develop memory disclosure exploit
3. Iteratively obtain memory pages online
4. Find ROP Gadgets
5. Develop an exploit
6. Exploit

Target Binary

Attacker

Victim

\x31\xc0\x50\x68...
History So Far ...

Direct code injection

NX / DEP

Code-reuse attacks (e.g., ROP)

ASLR

Exploiting fixed code section with ROP

Canary

w/ Memory Leak (e.g., JIT ROP)

Fine-grained ASLR
Question?