Stack Buffer Overflow

Wei Wang
Stack Buffer Overflow Vulnerability

- The first generation vulnerability
- Attacks to stack buffer overflow, and their defenses:
  - Basic stack smashing
    - General stack corruption
    - Corrupt return address
    - Code Injection
  - Off-by-One byte
  - Arc-injection or Return to libc
  - Return Oriented Programming (ROP)
Stack Smashing – Corrupting Stack

- Example: check_passwd1.c
- The stack before calling “gets”

```
<table>
<thead>
<tr>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved %ebp</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>'xxxx'</td>
</tr>
</tbody>
</table>
```

- pass
- buff
Stack Smashing – Corrupting Stack cont'd

• “gets” reads user input into “buff” without checking the length of user input

• User-input may over run the buffer, and modify the value of “pass”

• E.g., input is “AAAAA”, “pass” is corrupted, and attacker gains access

• We can write more to the stack to crash it – causing DoS
Stack Smashing – Corrupting Return Address

• Example: check_passwd2.c

• Attacking “pass” now is not very easy
  – The “pass” flag and buffer “buff” are separated
  – “pass” is set upon return of “check_password”
Stack Smashing – Corrupting Return Address cont'd

• However the buffer “buff” in function “check_password” is still unprotected

• The stack before calling “gets”

```
<table>
<thead>
<tr>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved %ebp</td>
</tr>
<tr>
<td>padding</td>
</tr>
<tr>
<td>padding</td>
</tr>
<tr>
<td>'xxxx'</td>
</tr>
</tbody>
</table>
```

buff
Stack Smashing – Corrupting Return Address cont'd

- We can still overflow the buffer and change the return address
- We can change the return address to bypass the check, and directly jump to the grant root privilege part
  - What is the instruction address we want to jump to?
  - Use objdump to find out
Stack Smashing – Corrupting Return Address cont'd

• But be careful with saved EBP, we don't want to modify it (otherwise the program will crash)
  – Luckily stack addresses remain constant from run to run (if stack protections are not applied)
  – Therefore, we can run the program first, use GDB to grab the value of the saved EBP
Stack Smashing – Corrupting Return Address cont'd

- What should the stack look like after the attack?

  - Any string you want. I use 12 “A”s; An “A” is 0x41 in ASCII table
  - Keep the saved EBP
  - Paddings, can be safely written over
  - buff

- Our input string should be exactly like these five bytes

- ASCII code vs keyboard typed character
Stack Smashing –
Corrupting Return Address cont'd

• How to input a ASCII character “0xf8”?

• Two methods to generate the attack string
  – Write a C program
    E.g., char a[] = “\xf8\xf5”;
  – Use perl
    E.g., perl -e 'print "\xf8\xf5";'

• How to pass the attack string to the program?
  – Save the attack string into a file, and use bash stdin redirect
    (Google bash redirect if you don't know)
  – Other programs may take input string differently
Stack Smashing –
Corrupting Return Address cont'd

- Now, let's try it out

- Extra considerations for GDB
  - GDB may shift stack address if program is started using “run” command,
  - GDB run check_passwd2 has a different saved EBP, wrong EBP value may crash the program
  - Stack address is affected by environment variables and command line parameters
  - The correct way to grab the saved EBP is to use GDB's attach process
Stack Smashing – Code Injection

- Attacker can also inject assembly codes into the stack
- Then attacker corrupt the return address to point to his injected code to execute
- With code injection, the attack can execute almost any codes he wants
Stack Smashing – Code Injection cont'd

- Example: `vulnerable_app.c`
- “`vulnerable_function`” has unprotected “`buff`”

![Diagram showing the structure of buffer with padding bytes for alignment and return address saved]
Stack Smashing – Code Injection cont’d

- Goal: inject code to print “xxxHACKEDxxx” on the screen
- Need to make system call: sys_write
- sys_write takes three parameters:
  - The length of the string to print, through EDX
  - The address of the string to print, through ECX
  - The id of the output device, through EBX
- To make a system call, system call number has to be specified in EAX
Stack Smashing – Code Injection cont'd

- Sample sys_write calling code:

```
.section .data
s:
  .string "Hello"

.section .text
.global main

main:
  movl $5, %edx  # length of "Hello"
  movl $s, %ecx  # address of "Hello"
  movl $1, %ebx  # 1 is stdout's device id
  movl $4, %eax  # 4 is sys_write's id.
  int $0x80      # soft-interrupt to OS Kernel
                   # to invoke system call
```

- We want inject similar code to stack
Stack Smashing – Code Injection cont'd

- The stack should look like this after injection:

- Need to inject machine code
  - use gcc to determine the correct machine code

- Injected code actually uses 23 bytes

```
mov $12, %edx
mov $0xbfffffff60, %ecx
mov $1, %ebx
mov $4, %eax
int $0x80
```

Address:

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbfffffff70</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0xbfffffff6c</td>
<td>&quot;Dxxx&quot;</td>
</tr>
<tr>
<td>0xbfffffff68</td>
<td>&quot;ACKE&quot;</td>
</tr>
<tr>
<td>0xbfffffff64</td>
<td>&quot;xxxH&quot;</td>
</tr>
<tr>
<td>0xbfffffff60</td>
<td>0xbfffffff70</td>
</tr>
<tr>
<td>0xbfffffff5c</td>
<td>0xbfffffff68</td>
</tr>
<tr>
<td>0xbfffffff58</td>
<td>'AAAA'</td>
</tr>
<tr>
<td>0xbfffffff54</td>
<td>'AAAA;'</td>
</tr>
<tr>
<td>0xbfffffff50</td>
<td>'AAAA'</td>
</tr>
<tr>
<td>0xbfffffff4c</td>
<td></td>
</tr>
</tbody>
</table>

Injected code

NULL-terminated string to print

Return address

Saved EBP

Padding

buff
Stack Smashing – Code Injection cont'd

- Note that code is injected from low address to high address, i.e., “mov $12, %edx” is at the bottom, “int $x80” is at the top in memory.

- Recall that the stack grows from high to low addresses, but instructions are saved from low to high addresses.

- Let's try it out.
Stack Smashing – Code Injection cont'd

- Execution environment difference makes it difficult to determine the exact location of inserted code
  - Environment variables, command line parameters may shift the beginning address of stack by a few bytes
  - Attackers can insert “nop”s to as a cushion

```
New Return Address

Saved EBP

Inserted Payload (code)

   nop
   nop
   ...
   nop
   nop
   jump to the middle of “nop”s
```

Un-protected buffer
Stack Smashing – Code Injection cont'd

- For the same example program, we can also injected code to create a new process
- E.g., to execute “/bin/cat /etc/passwd”
- We need to use system call “execve”
- The parameters are:
  \[\text{execve(char *filename, char *argv[], char * envp[])};\]
- Similarly, 1\(^{st}\) param in %ebx, 2\(^{nd}\) param in %ecx, 3\(^{rd}\) param in %ecx
- execve's system call id is 11
Stack Smashing – Code Injection cont'd

- The stack should look like this after injection:

```
Address: 0xbfffffe80  0x00000000
          0xbfffffe7c  0xbfffffe6c
          0xbfffffe78  0xbfffffe60
          0xbfffffe74  "swd\0"
          0xbfffffe70  "/pas"
          0xbfffffe6c  "/etc"
          0xbfffffe68  0x00000000
          0xbfffffe64  "/cat"
          0xbfffffe60  "/bin"
          0xbfffffe60  0xbfffffe84
          0xbfffffe5c  0xbfffffe68
          0xbfffffe58  'AAAA'
          0xbfffffe58  'AAAA'
          0xbfffffe54  'AAAA;'
          0xbfffffe50  'AAAA'
          0xbfffffe4c  buff
```

Arguments array: argv[]
Command-line parameter
NULL-terminated command
Return address
Saved EBP
Padding
Stack Smashing – Code Injection cont'd

- Stack cont'd

```asm
xor   %eax,%eax
mov   $0xb,%al
mov   $0xbfffffe60,%ebx
mov   $0xbfffffe78,%ecx
mov   $0xbfffffe80,%edx
int   $0x80
```

Address: 0xbfffe84

21 bytes of injected code
Stack Smashing – Code Injection cont'd

- The attackers usually insert code to start a shell, so that they can execute arbitrary commands.
- E.g., the attacker can attack a vulnerable web server, create a shell, and redirect the input to one of the server's network socket.
- If the injected payload (code) starts a shell, the payload is called “shellcode”.
- We can simply modify the above “execve” example to create a bash shell.
Summary of Stack Buffer Overflow and Stack Smashing

- Buffer overflow: a program writes more data to a buffer than the buffer's maximum size.
- The attack that deliberately causing buffer overflow is called *stack smashing*.
- An attack may corrupt data on the stack and cause program crash – Denial of Service attack.
Summary of Stack Buffer Overflow
and Stack Smashing cont'd

• An attack may corrupt local variables on the stack (e.g., check_passwd1)
  - Can become very power attack if there are pointers on the stack, e.g.,

```c
int *pointer;
int value;
Char buff[256];

gets(buff);    /* unprotected write to buff */
pointer = value; /* if both pointer and value */
              /* are corrupt, the attacker */
              /* can write any value to any */
              /* location */
```
Summary of Stack Buffer Overflow and Stack Smashing cont'd

- An more complex attack will try to overwrite return address to change execution flow
  - return address is one of the most popular target

- An even more complex attack can even inject code into stack to execute arbitrary code

- For complex stack smashing attacks, the attackers may need some knowledge of possible stack addresses to succeed