Reserve Engineering & Buffer Overflow Attacks

Tom Chothia
Computer Security, Lecture 17

Introduction
- A simplified, high-level view of buffer overflow attacks.
  - x86 architecture
  - overflows on the stack
- Some IDA examples.
- Exploiting buffer overflows using Metasploit

The x86 Architecture

The program code
Static variables, Strings, etc
Data in use
Registers e.g.
The Accumulator
Instruction point
Stack point

The Stack

The stack part of the memory is mostly ‘Last In, First Out’.
We can only write and read to the top of the stack.

The Stack

You write to the stack with push
You read and remove an item from the stack with pop

Screen shot, IDA
Common Pattern 1
Data is moved to a register, operation is call, result stored in memory location or register.

```
mov eax, [esp+1Ch]
add [esp+18h], eax
```

- Value at [esp+1Ch] is moved to register `eax`
- It is added to the value at [esp+18h]
- The results is stored at [esp+18h]

Common Pattern 2
Data is compared using “cmp” or “test”, then a jump is made based on the result.

```
cmp dword ptr [esp+1Ch], 3
jls short loc_80483DF
```

- Value at [esp+1Ch] compared to “3”
- If it is less than or equal to, the program jumps to location “loc_80483DF”
- Otherwise it continues to the next command.

Reasons For Reverse Engineering
- Analyse Malware
- Debug memory errors
- Analyse Legacy code
- Security Audit
  - Particularly detect backdoors e.g. D-link router: http://bit.ly/1e6VzzZ

Buffer Overflows
- In languages like C, you have to tell the compiler how to manage the memory.
  - This is hard.
- If you get it wrong, then an attacker can usually exploit this bug to make your application run arbitrary code.
- Countless worms, attacks against SQL servers, Web Servers, iPhone Jailbreak, SSH servers, …

USS Yorktown
US Navy Aegis missile cruiser
Dead in the water for 2 and a half hours due to a buffer overflow.

“Because of politics, some things are being forced on us that without political pressure we might not do, like Windows NT. If it were up to me I probably would not have used Windows NT in this particular application.”

Ron Redman, deputy technical director Aegis

Function calls
```
void main () {
PUSH <2>
function (1,2);
PUSH <1>
CALL <function>
}
```

- Arguments 1 & 2 are passed on the stack.
- The CALL instruction runs a function
  - EIP changed to point to code of function
  - Old EIP stored on stack
Function Calls, Simplified

PUSH <arg2>
PUSH <arg1>
CALL <function>

CALL writes the instruction point (EIP) onto the stack and then sets the EIP to equal the code for the function.

Later a return instruction restores the old EIP and the program continues

<table>
<thead>
<tr>
<th>Old EIP</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Stack</th>
</tr>
</thead>
</table>

Common Pattern 3

• Data is loaded onto the stack
• Function is called that uses these values,
• The result will be pointed to by eax
mov [esp+4], eax ; s2
mov dword ptr [esp], offset s1 ; 'exit'
call __strncmp

• Value in eax is moved to [esp+4]
• 'exit' is put on top of the stack
• String compare is called on these.
• The result will be returned in the eax register

<table>
<thead>
<tr>
<th>Old EIP</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Stack</th>
</tr>
</thead>
</table>

Buffer Overflows

• The instruction pointer controls which code executes,
• The instruction pointer is stored on the stack,
• I can write to the stack 😊

<table>
<thead>
<tr>
<th>Hello World</th>
<th>Old EIP</th>
<th>Hello World</th>
<th>Stack</th>
</tr>
</thead>
</table>

Buffers

1. Functions called with "Hello World" function (user input);
2. Arg and EIP written to stack function (char *str) {
3. Function runs char buffer[16];
4. Buffer allocated strcpy(str, buffer);
5. String copied

<table>
<thead>
<tr>
<th>Hello World</th>
<th>Old EIP</th>
<th>Hello World</th>
<th>Stack</th>
</tr>
</thead>
</table>

Buffer Overflow

If user input is more than 16 bytes?

function (user input);
function (char *str) {
 char buffer[16];
 strcpy(str, buffer);
}

<table>
<thead>
<tr>
<th>Old EIP</th>
<th>Stack</th>
</tr>
</thead>
</table>

Buffer Overflow

If user input is more than 16 bytes

function (user input);
function (char *str) {
 char buffer[16];
 strcpy(str, buffer);
}

<table>
<thead>
<tr>
<th>Old EIP</th>
<th>Stack</th>
</tr>
</thead>
</table>
Buffer Overflow

If user input is more than 16 bytes
1. Runs as before
2. But the string flows over the end of the buffer
3. EIP corrupted, segmentation fault

Once more, with malice
1. Runs as before
2. Attack send a very long message, ending with the address of some code that gives him a shell.
   - The attackers code could also be part of the message
3. The attackers value is copied over the old EIP
4. When the function returns the attacks code is run

Over Writing Other Values

Attacking the instruction pointer (EIP) is the most powerful technique. However, any memory value can be attacked:

• Over write arguments on the stack
  – e.g. change the parameters to a chmod call
• Overflows on the heap
  – e.g. rewrite a password in memory

Defenses

• Stack canaries:
  – values placed on the stack, which are later tested.
  – if the stack is over written then the value test will fail.
• Randomisation
  – Layout of the memory is randomised.
  – This makes it very hard for the attack to find the memory to overwrite or code to jump to.

For more information see the Secure Programming Module

Recommend Paper:

• “Smashing the Stack for Fun and Profit” Elias Levy (Aleph One)

A simple introduction to buffer overflows from the mid 90s.

Standard defences now stop the attacks in this paper, but it gives an excellent introduction.

Conclusion

Buffer overflows are the result of poor memory management in languages like C
– even the best programmers sometimes make mistakes.

Buffer overflow attacks exploit these to overwrite memory values.

This often lets an attack execute arbitrary code.