15-213 Recitation 5: Attack Lab

8 Feb 2016
Ralf Brown and the 15-213 staff
Agenda

- Reminders
- Stacks
- Attack Lab Overview
- Appendix: Arrays
- Appendix: Structs
- Appendix: More Assembly
Reminders

- Bomb lab is due **tomorrow**!
  - “But if you wait until the last minute, it only takes a minute!” - *NOT!*
  - Don't waste your grace days on this assignment
- Attack lab will be released **tomorrow**!
Stacks

- Last-In, First-Out
  - just like a stack of plates
  - pushes and pops to preserve registers must be in opposite order
- x86 stack grows down
  - lowest address is “top”
Stacks

- `%rsp` contains the address of the topmost element of the stack
- `pushq {value}` is same as `sub $8,%rsp` `mov {value}, (%rsp)`
- only constants and registers can be pushed
- `popq {reg}` is equivalent to `mov (%rsp), {reg}` `add $8, %rsp`
Stacks

- Stacks are useful for recursion
  - each call of the function gets a separate copy of arguments and local variables
  - the separate copy is needed for a limited time only – until the call returns
  - callee always ends before caller
- This is called “stack discipline”

- Stack space is allocated in Frames
  - state for a single instantiation of a function
Register Saving Conventions

- **Caller-Saved**
  - called function may do as it wishes with the register
  - must save/restore register in caller's stack frame if it still needs the value after a function call
  - registers used as function arguments are always caller-saved
  - result register %rax is also caller-saved

- **Callee-Saved**
  - if the function wants to change the register, it must save the original value in its stack frame and restore it before returning
  - the calling function may store temporary values across function calls in callee-saved registers
# x86-64 Register Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>callee saves</td>
</tr>
<tr>
<td>%rcx</td>
<td>argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>callee saves</td>
</tr>
<tr>
<td>%r8</td>
<td>argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>caller saves</td>
</tr>
<tr>
<td>%r11</td>
<td>caller saves</td>
</tr>
<tr>
<td>%r12</td>
<td>callee saves</td>
</tr>
<tr>
<td>%r13</td>
<td>callee saves</td>
</tr>
<tr>
<td>%r14</td>
<td>callee saves</td>
</tr>
<tr>
<td>%r15</td>
<td>callee saves</td>
</tr>
</tbody>
</table>
Local Variables

- Local variables which can't be stored in registers are stored on the stack
  - this includes arrays, structs, and anything which has its address taken
- Storage is allocated by simply decrementing %rsp the appropriate amount
- Cleanup consists of incrementing %rsp to free the stack space

```c
void foo() {
    int *v[6];
    ...
    return;
}
```
Stack Frames

- A frame can have many parts, but only those needed by the function are actually present
  - we consider the function args and return address to be part of the *caller's* frame because they are pushed *before* control transfers to the callee
- Function args 7+ are stored in order from lowest address to highest
  - equivalent to pushing in reverse order
  - access by offsetting from own stack frame
- When present, frame pointers let you traverse the chain of stack frames
Smashing the Stack

- What if a function has a bug that causes it to write beyond its own frame, possibly on some specific program input?
  - We call such overwrites “smashing the stack”
  - If the return address is overwritten, we could end up anywhere when the function returns
  - Classic article: “Smashing the Stack for Fun and Profit” by Aleph One (1996).
The goal of this lab is to figure out some input to be fed to a program to make it do things it was never designed for
- You'll do this by smashing the stack
- In the process, you will learn how to prevent and defend against such attacks
- READ THE WRITEUP! It shows you the techniques and helper programs you'll need to successfully exploit your target program.
Attack Lab: Code Injection

- You'll be given two compiled programs with a buffer-overflow vulnerability
- Part 1 has you smashing the stack to place code of your own on the stack to create the necessary conditions to call the target function
  - this means you'll have to craft the appropriate bytes in your attack string and then jump to them somehow once they're on the stack
Code Injection Example

- `strcpy` copies from input up to and including a NUL byte (`\0`)  
- if there are more than 31 bytes before the NUL, `strcpy` will write beyond the end of `buf`
- the next thing on the stack is the return address for `vuln()`...

```c
void vuln(char *input) {
    char buf[32];
    ...
    strcpy(buf, input);
    return;
}
```
Code Injection Example

- Simplest case: we just need to jump to a different address
  - fill the buffer with anything, then overwrite the return address
  - if we need to jump to 0xFFF15213 instead of the original 0xFEEDBEEF, a possible value for input is 0x41 (32 times) 0x13 0x52 0xF1 0xFF 0x00
  - remember byte order!
- After the \_strcpy\_, the stack will look as shown

```c
void vuln(char *input) {
    char buf[32];
    ...
    strcpy(buf, input);
    return;
}
```
Code Injection Example

- If we need to insert actual code on the stack, we must first figure out the value of %rsp!
  - run a copy of the program and break at the call to strcpy
- Put your exploit code in the buffer, and overwrite the return address with the address of the buffer
  - use gcc and objdump to generate the byte sequences for your injected code
  - code can't contain NUL bytes (newlines if exploiting gets)
Attack Lab: The Stack Has Protections!

- Modern CPUs allow data areas to be marked as non-executable
  - so you can't place exploit code on the stack and jump to it
- OSes use Address Space Layout Randomization to keep addresses unpredictable
  - an attack that works on one run might not on the next!
- Compiled code commonly uses “stack canaries”
  - unpredictable values stored between on-stack buffers and return address
  - if the value changes, the program is aborted rather than returning from the function (because the return address might have been corrupted)
In Part 2 of Attack Lab, the program has stack protections in place. If we can't build our code on the stack, we have to find snippets of existing code ("gadgets") that we can stitch together with an appropriate sequence of return addresses on the stack:

- hence the name ROP
- we just need to find the right byte sequences – those exact instruction sequences don't actually have to be a deliberate part of the program!
ROP: Finding Gadgets

- Look for byte sequences corresponding to an interesting instruction followed by `ret`
  - e.g. 0x59 0xC3 would be `pop %rbx; ret`
- Need to be creative in the instructions we execute
  - we probably can't find `mov $0x15213, %rax`
  - but we could stitch together the equivalent from
    - `pop %rbx`
    - `mov %rbx, %rax`
    - and a value of 0x15213 on the stack

Example inspired by content created by Professor David Brumley
ROP: The Exploit

- Find the gadgets
- Overflow the buffer to overwrite the return address and higher stack addresses with ROP instructions
  - each is a gadget address followed by any data the gadget pops off the stack

```c
void vuln(char *input) {
    char buf[32];
    ...
    strcpy(buf, input);
    return;
}
```

Example inspired by content created by Professor David Brumley
ROP: The Exploit Runs

- When `vuln()` returns, we jump to Address2 instead of the original caller
  - `%rax = ?`
  - `%rbx = ?`

Example inspired by content created by Professor David Brumley
ROP: The Exploit Runs

- The gadget at Address2 pops the top item off the stack
  - %rax = ?
  - %rbx = 0x00015213

```
Address1:
  mov %rbx, %rax
  ret
```

```
Address2:
  pop %rbx
  ret
```

```
next ROP address...
filler
Address1
0x00015213
Address2
filler
filler
filler
```

%rip %rsp
ROP: The Exploit Runs

- Next, the gadget at Address2 returns, which puts us at Address1
  - %rax = ?
  - %rbx = 0x00015213
ROP: The Exploit Runs

- The gadget at Address1 now copies %rbx into %rax
  - %rax = 0x00015213
  - %rbx = 0x00015213

Address1:
- mov %rbx, %rax
- ret

Address2:
- pop %rbx
- ret

y.a. ROP address...
next ROP address...
Address1
0x00015213
Address2
filler
filler
filler
filler
ROP: The Exploit Runs

- Finally, the gadget at Address1 returns, taking us to the next gadget
  - \%rax = 0x00015213
  - \%rbx = 0x00015213
- We've now executed the equivalent of
  - `mov $0x15213, %rax`
- (with a side effect)

```
Address1:
  mov %rbx, %rax
  ret

Address2:
  pop %rbx
  ret
```
Attack Lab Tools

- `gcc -c file.s`
  - convert the assembly code in `file.s` to object code in `file.o`
- `objdump -d file.o`
  - disassemble the code in file.o; shows the actual bytes for the instructions
- `./hex2raw`
  - convert hex codes into raw ASCII strings to pass to targets
- `gdb`
  - determine stack addresses
- `paper and pencil`
  - for drawing stack diagrams
More Useful GDB Commands

\texttt{x/[n]}i <address> \quad \text{disassemble } n \text{ instructions at } <\text{address}>
\texttt{b} <loc> if <cond> \quad \text{conditional breakpoint, stop only if } <\text{cond} \text{ true}
\texttt{cond} <bp> <cond> \quad \text{add condition to existing breakpoint } <\text{bp}>
\texttt{commands} <bp> \quad \text{execute commands when breakpoint } <\text{bp} \text{ hit}
\texttt{tbreak} <loc> \quad \text{set temporary breakpoint – auto-deletes when hit!}
\texttt{finish} \quad \text{run until current frame (function) returns, and print return value}
\texttt{layout asm} \quad \text{split the screen into separate disassembly and command windows}
\texttt{layout reg} \quad \text{show register window as well (after layout asm)
If You Get Stuck

- Please read the writeup. *Please read the writeup. Please read the writeup. Please read the writeup!*
- CS:APP Chapter 3
- View lecture notes and course FAQ at [http://www.cs.cmu.edu/~213](http://www.cs.cmu.edu/~213)
- Office hours Sunday through Thursday 5:00-9:00pm in WeH 5207
- Post a **private** question on Piazza
- `man gdb`, gdb's `help` command
Remember...
Appendix: Arrays

- In C, the name of an array is interpreted as a pointer to the first element
  - A is the same as &A[0]
- Array subscripting is just a synonym for pointer arithmetic:

- This translates almost directly into assembly. x = A[5] becomes
  
  ```
  mov $5, %rax
  mov {address of A}, %rbx
  mov (%rbx, %rax, 4), %rdx
  ```

- We simply scale the index by the size of an element and add that to the starting address
Two-dimensional Arrays

- Arrays elements can themselves be arrays
- As with one-dimensional arrays, the elements are stored in order in memory
- C only supports compile-time sized multi-dimensional arrays – you need to compute the corresponding index as if the array were one-dimensional for run-time sizing

```c
int A[3][2] = {
    { 0, 1 },
    { 10, 11 },
    { 20, 21 } 
};
```

For a MxN array, `&A[i][j] = A + i*N + j`
Appendix: Structs

- Structures are a way to bundle together related data/variables.
- Elements of a structure may be of any type, including pointers, arrays and other structures (no recursive regress allowed!)
- Structs can be the elements of an array.
- Access parts of the structure by name, rather than by index as for arrays.

```c
struct info {
    int whole_num;
    double float_num;
    char string[9];
} S[5];
/* init 2\textsuperscript{nd} array element */
S[1].whole_num = 15213;
S[1].float_num = 15.213;
strcpy(S[1].string,"15-213");
```
Structs

- Structures are a way to bundle together related data/variables.

- Elements of a structure can be of any type, including pointers, arrays and other structures (no recursive regress allowed!)

- Structs can be the elements of an array.

- Access parts of the structure by name, rather than by index as for arrays.

```c
struct info {
    int whole_num;
    double float_num;
    char string[9];
} S[5];
/* init 2nd array element */
S[1].whole_num = 15213;
S[1].float_num = 15.213;
strcpy(S[1].string,"15-213");
```
The type name of a struct can be used in future variable declarations – given the declaration on the previous slide, we can now say:

```c
struct info T;
```

This declares variable `T` to be of the same type as the elements of array `S`.

The type name is optional when declaring a variable – that creates an *anonymous* structure type.

You can also declare a structure type without declaring any variables:

```c
struct st { int m1; float m2; };
```
Structs: Memory Layout

- Struct members are placed a multiple of their own size from the start of the struct
  - some architectures require such alignment to even access the member
  - x86 doesn't care, but will be slower whenever misalignment causes two memory accesses
- Minimize size by putting largest members first
  - array elements count as individual members

```
struct s1 {
    int i;
    double *p;
    char s[9];
};
```

```c
struct s2 {
    double *p;
    int i;
    char s[9];
};
```

- Minimize size by putting largest members first

```
struct s1 {
    int i;
    double *p;
    char s[9];
};
```

```
struct s2 {
    double *p;
    int i;
    char s[9];
};
```

sizeof s1 = 32

- Minimize size by putting largest members first

```
struct s1 {
    int i;
    double *p;
    char s[9];
};
```

```
struct s2 {
    double *p;
    int i;
    char s[9];
};
```

sizeof s2 = 24

- Minimize size by putting largest members first

```
struct s1 {
    int i;
    double *p;
    char s[9];
};
```

```
struct s2 {
    double *p;
    int i;
    char s[9];
};
```

sizeof s1 = 32

- Minimize size by putting largest members first

```
struct s1 {
    int i;
    double *p;
    char s[9];
};
```

```
struct s2 {
    double *p;
    int i;
    char s[9];
};
```

sizeof s2 = 24
Arrays of Structs

- Requiring struct sizes to be a multiple of their strictest alignment supports arrays
- Otherwise, not all members would be aligned in every array element

Diagram showing alignment issues with misaligned data.
Appendix: More Assembly

- Some instructions you may encounter
  - `cltq` (“Convert Long To Quad”) -- sign-extend %eax into %rax
  - `cmovX` (“Conditional Move”) -- executes move only if condition X is true
  - `movzbl` (“MOVe w/ Zero-extension, Byte to Long”)
  - `nop` (“No Operation”) -- do nothing
  - `nopl` (“No Operation, Long”) -- multi-byte instruction that does nothing
    - used to align function start addresses to a multiple of 16 bytes
  - `reptz retq` – see Recitation 4
  - `mov %fs:0x28, %rax`
    - beyond the scope of the course; %fs is a *segment register*, which here is used to implement thread-local storage