CSE 127 Discussion 3

Brendon Chen
Some slides adapted from Stefan Savage and SP19’s CSE 127
This session is being recorded
Reminders

- PA2 due next Tuesday, April 27 at 11:00:00 AM PDT
- Midterm next Thursday, April 29
Today’s Agenda

1. Review
   a. Stack Canaries
   b. Data Execution Prevention (DEP)
   c. Address Space Layout Randomization (ASLR)
   d. Return to libc
   e. Return Oriented Programming (ROP)
   f. Control Flow Integrity (CFI)
   g. System Security I

2. PA2

3. Open Office Hours
Review
## Stack Canaries

- **Mitigation that detects overwrite of stack into control data**
- **Limitations**
  - Do not protect from non-sequential overwrites
    - Local variables
  - Do not prevent the overwrite
- **Bypass**
  - Terminator canaries are not impossible to insert
  - Possible to learn the canary value
  - Information leaks
- **Considered essential mitigation on modern systems**
  - Offer significant value for relatively little cost

<table>
<thead>
<tr>
<th>argv[1]</th>
<th>saved ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbbbbbbbbbbb</td>
<td>saved ebp</td>
</tr>
<tr>
<td>0xaaaaaaaaaa</td>
<td>canary</td>
</tr>
<tr>
<td>saved ebp</td>
<td>0xdeadbeef</td>
</tr>
<tr>
<td>buf[0-3]</td>
<td></td>
</tr>
</tbody>
</table>

```
%ebp →
<table>
<thead>
<tr>
<th>canary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

%esp →
```
Data Execution Prevention (DEP)

- Mitigation that marks stack and heap as non-executable (hardware support)
- Mark all pages either writable or executable, but not both
  - Stack and heap are writable, but not executable
  - Code is executable, but not writeable
  - Also known as W^X (Write XOR eXecute)
- Bypass
  - What if pages need to be both writable and executable?
  - What if there is useful executable code you can repurpose?
  - Attackers can still execute arbitrary code even without injecting malicious code
Address Space Layout Randomization (ASLR)

- Mitigation that randomizes location of key data structures (e.g., stack and heap)
- Requires compiler, linker, and loader support
- Side Effects
  - Increases code size and performance overhead
  - Random number generator dependency (like candaries)
  - Potential load time impact for relocation (shared libraries/DLLs)
- Bypass
  - Longer NOP sleds
  - Information leaks/guessing
  - Heap Spraying
Return to libc

- Transfer control to address of system() in libc
- Setup stack frame to look like a normal call to system()
- Many different variants
- Other things attackers do by calling available functions
  - Move shellcode to unprotected memory
  - Change permissions on stack pages (mprotect())
  - etc.
Return Oriented Programming (ROP)

- Idea: make complex shellcode out of existing application code
  - ROP Gadgets: code sequences ending in ret instruction
  - Stitch together arbitrary programs out of code gadgets already present in the target library
  - x86 has variable-length instructions == more options!
- Stack pointer acts as instruction pointer
- Manually stitching gadgets together gets tricky
  - Automation!
%edx = v₁

mov v₁, %edx
Control Flow Integrity (CFI)

- **Idea**: restrict control flow to legitimate paths
  - i.e., ensure that jumps, calls, and returns can only go to allowed target destinations
- **Direct control flow transfer**
  - Advancing to next sequential instruction
  - Jumping to (or calling a function at) an address hard-coded in the instruction
  - These are static in code, so assume attacker can’t control
- **Indirect control flow transfer**
  - Jumping to (or calling a function at) an address in register or memory
  - Forward path: indirect calls and branches (e.g., a function you are calling)
  - Reverse path: return addresses on the stack (returning from a called function)
void sort2(int a[], int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
System Security I

- **Process isolation**
  - Hardware support (MMU)
  - Provides separate address spaces to different processes
  - Control modes of access to memory (i.e., R, W, X)

- **User/Kernel Privilege Separation**
  - Processor privilege modes use to limit access to sensitive instructions/memory
  - Careful checking of syscall interface from user processes
  - Map Kernel into all process address spaces to make memory calls fast (problematic)

- **Virtual machines**
  - Same idea, but add another level of isolation (hypervisor -> OS -> process)
Little Endianness

```c
int arr[2];
arr[0] = 0x12345678;
arr[1] = 0xAABBCCDD;
```

```
| char chrs[8] = {10, 20, 30, 40, 50, 60, 70, 80}; |
|----------------|----------------|----------------|----------------|
| 0x78            | 0x56            | 0x34            | 0x12           |
| 0xDD            | 0xCC            | 0xBB            | 0xAA           |
| 10              | 20              | 30              | 40             |
| 50              | 60              | 70              | 80             |
```

Credit: CSE 127 SP19
```c
int bar(char *arg, char *out)
{
    strcpy(out, arg);
    return 0;
}

int foo(char *argv[])
{
    char buf[768];
    bar(argv[1], buf);
}

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

ESP -> [foo] <padding>
&buf -> [foo] buf

Low

[foo] <saved reg>

EBP -> [foo] main's ebp
EBP+4 -> [foo] return addr to main
(Main's EBP) -> [foo] argv

High

[main] < saved register >
[main] < saved register >
[main] argv
[main] argv
```c
void nstrncpy(char *out, int outl, char *in)
{
    int i, len;
    len = strlen(in);
    if (len > outl)
        len = outl;
    for (i = 0; i <= len; i++)
        out[i] = in[i];
}

void bar(char *arg)
{
    char buf[107];
    nstrncpy(buf, sizeof(buf, arg));
}

void foo(char *argv[])
{
    bar(argv[1]);
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target2: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    foo(argv);
    return 0;
}
```

```c
0xbffff720
[bar] <padding>

0xbffff790
[bar] buf
[bar] foo's ebp
[bar] return addr to foo
[bar] arg
[foo] main's ebp
[foo] return addr to main
[foo] argv
main
```
void nstrncpy(char *out, int outl, char *in)
{
    int i, len;
    len = strlen(in);
    if (len > outl)
        len = outl;
    for (i = 0; i <= len; i++)
        out[i] = in[i];
}

void bar(char *arg)
{
    char buf[107];
    nstrncpy(buf, sizeof buf, arg);
}

void foo(char *argv[])
{
    bar(argv[1]);
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        printf(stderr, "target2: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    foo(argv);
    return 0;
}

sploit2 (credit: CSE 127 SP19)
- Things to pay attention to
  - `strtoul()`
  - `count < 635`
  - `count * sizeof(struct widget_t)`

- Think of
  - Signed vs. unsigned ints/longs
  - Multiplication vs. shift
    - $9 \times 32 = 9 \ll 5$
    - $4160749577 \times 32 = 4160749577 \ll 5$
    - All equal to 288

```c
struct widget_t {
    double x[1];
};

int foo(char *in, int count) {
    struct widget_t buf[635];

    if (count < 635)
        memcpy(buf, in, count * sizeof(struct widget_t));

    return 0;
}

int main(int argc, char *argv[]) {
    int count;
    char *in;

    if (argc != 2) {
        fprintf(stderr, "target3: argc ! = 2\n");
        exit(EXIT_FAILURE);
    }

    /* format of argv[1] is as follows: */
    /* - a count, encoded as a decimal number in ASCII */
    /* - a comma (",") */
    /* - the remainder of the data, treated as an array */
    /* of struct widget_t */
    /* */

    count = (int)strtoul(argv[1], &in, 10);
    if (*in != ',') {
        fprintf(stderr, "target3: argument format is [count],[data]\n");
        exit(EXIT_FAILURE);
    }

    in++; /* advance one byte, past the comma */
    foo(in, count);
    return 0;
}
```
sploit4 Supplemental Slides
Open Office Hours