Low-level Security

Low-level Software Security: Attacks and Defenses
and
Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns

by
Eric Hardisty
Low-level Security

Become a Dark Lord of Assembly Hacking
A Quick Intro...

• What is low-level security?
  – The portion of computer security that involves exploits at the level of compiled and loaded source code, or more generally, at the boundary of any language translation. It can be highly dependent on architecture.

• Why do we care?
  – C/C++ are the most vulnerable and also the most used.
  – Lots of opportunity for exploitation and no “catch-all” solution.

• Why is it difficult/interesting?
  – “Low-level attacks may be possible whenever software is translated from a higher-level language into a lower-level language without a guarantee that this translation preserves the higher-level abstractions.”
Organization

• Attacks:
  - Smashing the Stack (for fun and profit)
  - Function pointer corruption
  - Return-to-libc, a.k.a. arc injection
  - Data-only corruption

• Selected Defenses:
  - Stack canaries
  - Rearranging the stack
  - Address space layout randomization (ASLR)
Smashing the Stack

int is_file_foobar( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}

1. Buffer overflow occurs here with two=“asdfsdfasdfsdf”
2. Better yet, set two so that this becomes 0x0012ff48, and...

3. Put payload function call address as the second char of tmp, since that is what
   the program counter will be as soon as the function returns.
4. Bask in the power of the Dark Side!

address    content
0x0012ff5c 0x00353037 ; argument two pointer
0x0012ff58 0x0035302f ; argument one pointer
0x0012ff54 ; return address
0x0012ff50 0x61666473 ; saved base pointer 's' 'd' 'f' 'a'
0x0012ff4c 0x61666473 ; tmp continues 's' 'd' 'f' 'a'
0x0012ff48 0x61666473 ; tmp continues 's' 'd' 'f' 'a'
0x0012ff44 0x6162f3a ; tmp continues 'i' '/' '/' 'a'
0x0012ff40 0x656c6966 ; tmp array: 'f' 'i' 'l' 'e'
Smashing the Stack

- Constraints
  - Relies on the hardware being willing to execute the stack contents as machine code.
  - Whatever the attacker wants to put into the stack as the overflow cannot contain zeros.

- Variations
  - Clobber the base address.
  - Clobber an exception handler pointer, then cause the exception you need.
Function Pointer Corruption

```c
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*, char*);
} vulnerable;

int is_filefoobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file:///foobar" );
}
```

<table>
<thead>
<tr>
<th>buff (char array at start of the struct)</th>
<th>cmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>address: 0x00353068 0x0035306c 0x00353070 0x00353074</td>
<td>0x00353078</td>
</tr>
<tr>
<td>content: Oxfeeb2ecd Ox111111111 0x111111111 0x111111111</td>
<td>0x00353068</td>
</tr>
</tbody>
</table>

1. Place the exploit instruction in the first byte of buff.
2. Overflow buff to place the start address of the struct into the cmp function pointer.
3. Attacker’s code is invoked at the return statement.
Function Pointer Corruption

• Constraints
  - Determining the address of the heap memory to be corrupted.
  - The address the attacker needs to use cannot contain any zeros!

• Variations
  - Two consecutive objects on the heap. One object overflows a function pointer in the other.
  - Heap metadata corruption.
  - *Implementation of subtype polymorphism in C++ (virtual function tables) makes it particularly vulnerable.* Yikes!
Return-to-libc (Arc Injection)

- Differs from the previous two attacks in that no new code is injected by the attacker.

- Attacker causes the execution of one or more library functions which were unreachable in the original source code.

- Although the library function to be executed is dead in the target source code, it’s a shared library so it cannot be removed from the process address space.

- Attackers can use this indirect code injection if the system does not allow execution of data on the stack.
Return-to-libc (Arc Injection)

int median(int* data, int len, void* cmp)
{
    // must have 0 < len <= MAX_INTS
    int tmp[MAX_INTS];
    memcpy(tmp, data, len*sizeof(int));  // copy the input integers
    qsort(tmp, len, sizeof(int), cmp);    // sort the local copy
    return tmp[len/2];                    // median is in the middle
}

1. Find a trampoline at a known address somewhere in the system libraries.

2. Overflow tmp to place that address in cmp.

3. Note: the ebx register contains the address of tmp.
Return-to-libc (Arc Injection)

- First instruction changes the stack to point to the first address of the tmp array (the one the attacker just overflowed.)
- Throw away the first element in tmp.
- Return statement invokes the payload.
- The attacker chooses the stack contents to call arbitrary library functions with arbitrary arguments as the stack is unwound.

```c
// call a function to allocate writable, executable memory at 0x70000000
VirtualAlloc(0x70000000, 0x1000, 0x3000, 0x40); // function at 0x7c809a51

// call a function to write the four-byte attack payload to 0x70000000
InterlockedExchange(0x70000000, 0xfeeb2ecd); // function at 0x7c80978e

// invoke the four bytes of attack payload machine code
((void (*)(void))(0x70000000))(); // payload at 0x70000000
```
Return-to-libc (Arc Injection)

- Constraints
  - The attacker must know the addresses of the specific machine instructions useful for the attack. This can be tough as the addresses can change with different versions.

- Variations
  - Target the “system” system call.

- What makes this particularly insidious?
  - We can’t eliminate all the library code that could be useful to the attacker.
  - “[F]or x86 Linux software... it is practical for elaborate jump-to-libc attacks to perform arbitrary functionality while executing only machine-code found embedded within other instructions.”
void run_command_with_argument( pairs* data, int offset, int value )
{
    // must have offset be a valid index into data
    char cmd[MAX_LEN];
    data[offset].argument = value;
    {
        char valuestring[MAX_LEN];
        itoa( value, valuestring, 10 );
        strcpy( cmd, getenv("SAFECOMMAND") );
        strcat( cmd, " " );
        strcat( cmd, valuestring );
    }
    data[offset].result = system( cmd );
}

• Attacker chooses offset and value such that the address of their attack command gets placed into the first entry in the environment table.

• system( cmd ) then invokes their attack.
Data-only Corruption

• Constraints
  - Software is likely to allow only certain data to be corrupted and potentially only in certain ways.
  - Attacker constrained by the behavior of the target software when given arbitrary input.

• Variations
  - Not a common attack. Usually used as the first step in a more powerful attack, like jump-to-libc.
Organization

- Attacks:
  - Smashing the Stack (for fun and profit)
  - Function pointer corruption
  - Return-to-libc, a.k.a. arc injection
  - Data-only corruption

- Selected Defenses:
  - Stack canaries
  - Rearranging the stack
  - Address space layout randomization (ASLR)
Stack Canaries

- Place a “tell-tale” value above the stack buffers.
- Check for its integrity to determine if the stack buffer has been overflowed.
- For increased security, use a canary value that’s unknown to the attacker.
- Very cheap (computationally), but only defends against smashing the stack, not against the more advanced attacks.
Rearranging the Stack

- Place copies of sensitive data on the top of the stack!
- Cannot be overrun by the buffers below it on the stack.
- Offers no protection against heap smashing or data-only attacks, but it is efficient and effective against stack smashing and return-to-libc attacks.
ASLR

• C/C++ languages do not specify the process memory layout. This can be used as a defense.

• Attacks (so far) depend on the concrete addresses used during execution. ASLR deprives the attacker of these addresses by randomly changing the address space of the software.

• Employed in various forms by Linux, OS X 10.5, and by Windows Vista.

• A basic form of pointer encryption, but much less expensive.

• System depends on a secret, so it is no defense against an attacker that can read memory. It should be assumed that ASLR is compromised once an attacker has the ability to execute even the smallest amount of code of their choice.

• Fortunately, ASLR makes it very tough to do the above!
Hmm...questions, you have.