CS326 – Systems Security

Lecture 11

Control-flow Attacks

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy
Functions

• Software is composed by several functions
  – main(), printf(), malloc(), create_user(), etc.

• Functions allow code re-use
  – Whenever you want to display a message you simply call printf

• In a program a function can call a function, and then another function
  – All this function chaining is called control flow
The life of a function

• Whenever a function is called, the control flow of the program is changed
  – We need to do this transparently
  – Once the function is finished the control flow should be resumed

• Functions may take arguments
• Functions may return data
• Functions may create local data
Vocabulary

• When a function \texttt{foo} is called
  – \texttt{foo} is the callee
  – The address that called the function is called \texttt{call site} (or caller)
The stack

• Functions need memory for their work
  – This is the stack

• This memory is for short lived data
  – Once the function is finished we can get rid of the data involved

• Architecture dependent
  – The main idea does not change

• The stack may hold several things
  – Function arguments, the return address, the old frame pointer, local arguments
Stack of Intel (32-bit)

- The stack grows from higher-memory addresses to lower-memory addresses
  - It is like the stack is flipped upside down
- The top of the stack is always kept in a hardware register (esp)
- Each function creates a new stack frame upon executing
  - A virtual portion inside the stack
  - The stack frame is destroyed once the function is finished
- The top of the stack frame is kept in a hardware register (ebp)
Stack insertion

- `push %eax`
  
  `sub 0x4, %esp`
  
  `mov %eax, (%esp)`

(0xffffffa4b6) %esp → top of stack

High-memory addresses

Low-memory addresses
Stack insertion

- `push %eax`
  `sub 0x4, %esp`
  `mov %eax, (%esp)`

```
(0xfffffa4b2) %esp
old top of stack
contents of %eax
```

High-memory addresses

Low-memory addresses
Stack deletion

- `pop %eax`
  - `mov %(%esp), %eax`
  - `add 0x4, %esp`

```
(0xffffffffb2) %esp
```

```
0xfff
```

High-memory addresses

Low-memory addresses

Top of stack (0x42)
Stack deletion

• `pop %eax`
  
  `mov %(%esp), %eax`
  
  `add 0x4, %esp`

---

(0xfffffa4b6) %esp %eax holds 0x42

new top (0xff)

High-memory addresses

Low-memory addresses
## Stack frame in Intel 32-bit

<table>
<thead>
<tr>
<th>High-memory addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function arguments</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>%ebp (frame pointer)</td>
</tr>
<tr>
<td>Environment (e.g., argv)</td>
</tr>
<tr>
<td>Local variables</td>
</tr>
</tbody>
</table>

| Low-memory addresses |

- 11
Endianness

- Assume the 32-bit word: 0xA0B0C0D
- Two possible ways to store it in memory

### Little Endian (Intel)
- 0xD 0xC 0xB 0xA
- Low Mem → High Mem

### Big Endian (Motorola)
- 0xA 0xB 0xC 0xD
- Low Mem → High Mem
Example
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int password_valid = 0;

void authenticate_root(char *passwd) {
    unsigned long marker = 0xdeadbeef;
    char password[16];

    strcpy(password, passwd);

    fprintf(stderr, "Validating password: %s\n", password);

    if (!strcmp(password, "e5ce4db216329f4f"))
        password_valid = marker;
}

int main(int argc, char *argv[]) {
    authenticate_root(argv[1]);

    if (password_valid != 0) {
        printf("Welcome administrator.\n");
    } else {
        printf("Access denied.\n");
    }

    return 1;
}
```
Normal use

```bash
elathan@l64:~/ucy/epl326/8$ gcc -Wall -m32 -no-pie -fno-pic -fno-stack-protector stack-smash.c -o stack-smash
elathan@l64:~/ucy/epl326/8$ ./stack-smash AA
Validating password: AA
Access denied.
elathan@l64:~/ucy/epl326/8$ ./stack-smash e5ce4db216329f4f
Validating password: e5ce4db216329f4f
Welcome administrator.
elathan@l64:~/ucy/epl326/8$
elathan@l64:~/ucy/epl326/8$ printf "AA" | xargs ./stack-smash
Validating password: AA
Access denied.
elathan@l64:~/ucy/epl326/8$ printf "e5ce4db216329f4f" | xargs ./stack-smash
Validating password: e5ce4db216329f4f
Welcome administrator.
elathan@l64:~/ucy/epl326/8$
```
elathan@l64:~/ucy/epl326/8$ gdb ./stack-smash
GNU gdb (Ubuntu 7.12.50.20170314-0ubuntu1) 7.12.50.20170314-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
Find the GDB manual and other documentation resources online at:
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stack-smash...(no debugging symbols found)...done.
(gdb) b authenticate_root
Breakpoint 1 at 0x80484d1
(gdb) r AAAA
Starting program: /home/elathan/ucy/epl326/8/stack-smash AAAA

Breakpoint 1, 0x080484d1 in authenticate_root ()
(gdb)
Breakpoint 1, 0x080484d1 in authenticate_root()
(gdb) disas
Dump of assembler code for function authenticate_root:
  0x080484cb <+0>:  push   %ebp
  0x080484cc <+1>:  mov    %esp,%ebp
  0x080484ce <+3>:  sub    $0x28,%esp
  => 0x080484d1 <+6>:  movl   $0xdeadbeef,-0xc(%ebp)
  0x080484d8 <+13>: sub    $0x8,%esp
  0x080484db <+16>: pushl   0x8(%ebp)
  0x080484de <+19>: lea    -0x1c(%ebp),%eax
  0x080484e1 <+22>: push   %eax
  0x080484e2 <+23>: call   0x8048380 <strcpy@plt>
  0x080484e7 <+28>: add    $0x10,%esp
  0x080484ea <+31>: mov    0x804a028,%eax
  0x080484ef <+36>: sub    $0x4,%esp
  0x080484f2 <+39>: lea    -0x1c(%ebp),%edx
  0x080484f5 <+42>: push   %edx
  0x080484f6 <+43>: push   $0x8048610
  0x080484f8 <+48>: push   %eax
  0x080484fc <+49>: call   0x80483b0 <printf@plt>
  0x08048501 <+54>: add    $0x10,%esp
  0x08048504 <+57>: sub    $0x8,%esp
  0x08048507 <+60>: push   $0x8048629
  0x0804850c <+65>: lea    -0x1c(%ebp),%eax
  0x0804850f <+68>: push   %eax
  0x08048510 <+69>: call   0x8048370 <strcmp@plt>
  0x08048515 <+74>: add    $0x10,%esp
  0x08048518 <+77>: test   %eax,%eax
  0x0804851a <+79>: jne    0x8048524 <authenticate_root+89>
  0x0804851c <+81>: mov    -0xc(%ebp),%eax
  0x0804851f <+84>: mov    %eax,0x804a030
  0x08048524 <+89>: nop
  0x08048525 <+90>: leave
  0x08048526 <+91>: ret

End of assembler dump.
(gdb)
Control-flow Attacks

• The memory of the process contains control data
• In our example, this is the return address stored in the stack
• Control data dictate the flow of the program
• Overwriting control data hijacks the control flow
• Overwriting is possible, since control data are co-located with other buffers that can be overwritten due to program’s vulnerabilities
Was the attack perfect?

• Not the best we could do
  – Stack was not handled correctly
  – The program crashes at the end
• Easy to carry out
  – Just change the value of the return address
  – Goal achieved (although, dirty)
Further reading

• http://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/
• http://10kstudents.eu