Secure Programming Lecture 4: Memory Corruption II (Stack & Heap Overflows)

David Aspinall

Informatics @ Edinburgh

Classifying memory corruption errors

Spatial memory errors

An error happens because memory access goes outside the region of memory that a data item is intended to occupy.

Temporal memory errors

An error happens because memory access happens in some region of memory that the program *ought not currently* have access to.

This lecture focuses on spatial errors.

Buffer overflow

Buffer overflow is a common programming error.

- Simple cause:
  - putting *m* bytes into a buffer of size *n*, for *m*>*n*
  - corrupts the surrounding memory
- Simple fix:
  - check size of data before/when writing

Overflow *exploits*, where corruption performs something specific the attacker wants, can be very complex.

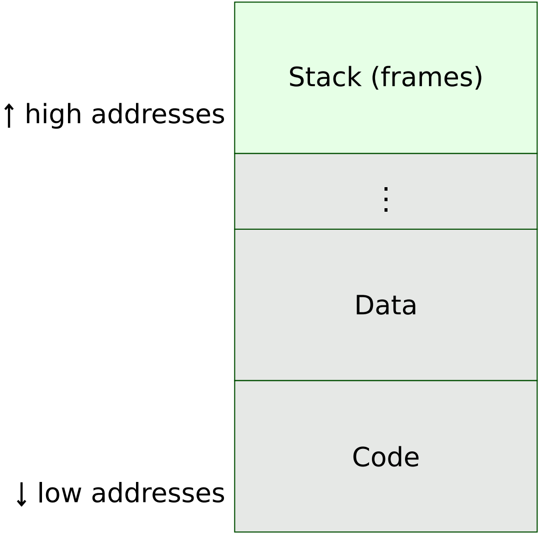
We'll study examples to explain how devastating overflows can be, looking at simple (mainly historical) **stack overflows** and **heap overflows**.

Examples will use Linux/x86 to demonstrate; principles are similar on other OSes/architectures.

Outline

- Stack variable corruption
- Executable code exploits
  - Shellcode
  - Redirecting execution
- Heap overflows
  - Specific heap attacks
  - General heap attacks
- Summary

How the stack works (reminder)



↑ high addresses

Stack (frames)

⋮

Data

Code

↓ low addresses

Memory

Corrupting stack variables

Local variables are put close together on the stack.

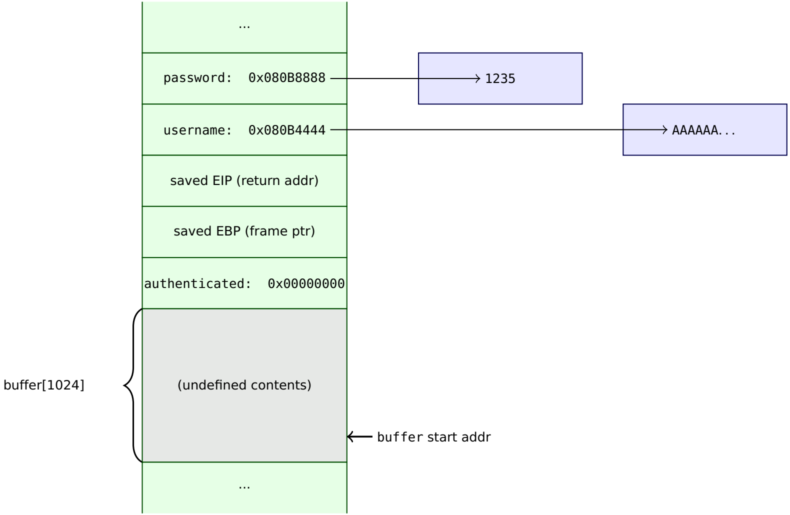
- If a stray write goes beyond the size of one variable
- . . . it can corrupt another

Application scenario

```c
int authenticate(char *username, char *password) {

  int authenticated; // flag, non-zero if authenticated
  char buffer[1024]; // buffer for log message

  authenticated = verify_password(username, password);

  if (authenticated == 0) {
    sprintf(buffer,
        "Incorrect password for user %s\n",
        username);
    log("%s",buffer);
  }
  return authenticated;
}
```

- Vulnerability in `authenticate()` call to `sprintf()`.
- If the `username` is longer than 995 bytes, data will be written past the end of the `buffer`.

Possible stack frame before exploit



...

password: 0x080B8888

1235

username: 0x080B4444

AAAAAA...

saved EIP (return addr)

saved EBP (frame ptr)

authenticated: 0x00000000

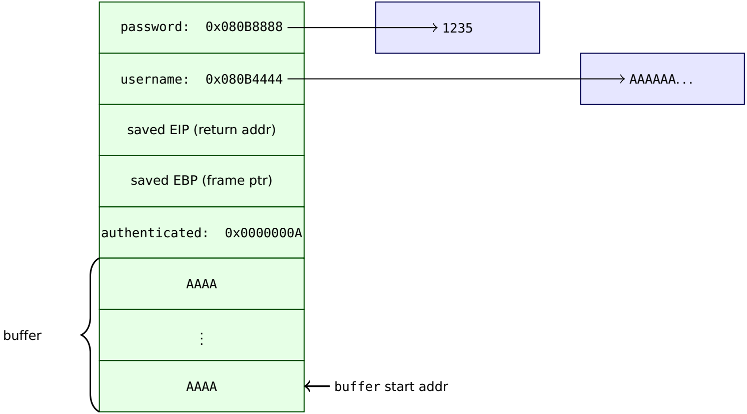
buffer[1024]

(undefined contents)

buffer start addr

...

Stack frame after exploit



- If `username` is >995 letters long, `authenticated` is corrupted and may be set to non-zero.
- E.g., char 1024=‘`\n`’, the low byte becomes 10.

Local variable corruption remarks

Tricky in practice:

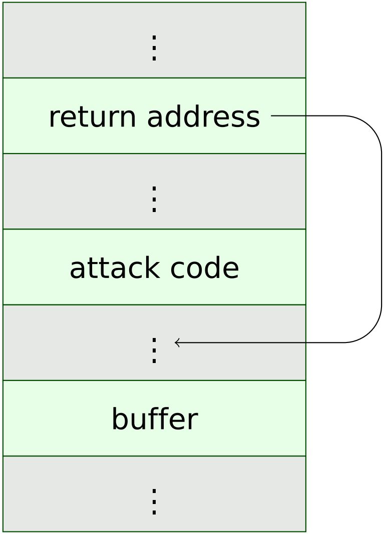
- location of variables may not be known
    - memory addresses can vary between invocations
    - C standards don't specify stack layout
    - compiler moves things around, optimises layout
- effect depends on behaviour of application code

A more predictable, *general* attack works by corrupting the fixed information in every stack frame: the frame pointer and return address.

Outline

- Stack variable corruption
- Executable code exploits
  - Shellcode
  - Redirecting execution
- Heap overflows
  - Specific heap attacks
  - General heap attacks
- Summary

Classic stack overflow exploit



The malicious argument overwrites all of the space allocated for the buffer, all the way to the return address location.

The return address is altered to point back into the stack, somewhere before the attack code.

Typically, the attack code executes a shell.

Attacker controlled execution

By over-writing the return address, the attacker may either:

1. set it to point to some known piece of the application code, or code inside a shared library, which achieves something useful, or
2. supply his/her own code somewhere in memory, which may do anything, and arrange to call that.

The second option is the most general and powerful.

How does it work?

Arbitrary code exploit

The attacker takes these steps:

1. **write code useful for an attacker**
2. store executable code somewhere in memory
3. use stack overflow to direct execution there

The attack code is known as **shellcode**. Typically, it launches a shell or network connection.

Shellcode is ideally:

- small and self-contained
- position independent
- free of ASCII NUL (0x00) characters

Question. Why?

Arbitrary code exploit

1. **write code useful for an attack**
2. store executable code somewhere in memory
3. use stack overflow to direct execution there

Outline

- Stack variable corruption

- Executable code exploits
  - Shellcode
  - Redirecting execution

- Heap overflows
  - Specific heap attacks
  - General heap attacks

- Summary

Building shellcode

Consider spawning a shell in Unix. The code looks like this:

```c
#include <unistd.h>
...
char *args[] = { "/bin/sh", NULL };
execve("/bin/sh", args, NULL)
```

- `execve()` is part of the Standard C Library, `libc`
- it starts a process with the given name and argument list and the environment as the third parameter.

We want to write (relocatable) assembly code which does the same thing: constructing the argument lists and then invoking the execve function.

Invoking system calls

To execute a *library* function, the code would need to find the location of the function.

- for a dynamically loaded library, this requires ensuring it is loaded into memory, negotiating with the linker
- this would need quite a bit of assembly code

It is easier to make a *system call* directly to the operating system.

- luckily, `execve()` is a library call which corresponds exactly to a system call.

Invoking system calls

Linux system calls (32 bit x86) operate like this:

- Store parameters in registers EBX, ECX, . . .
- Put the desired system call number into AL
- Use the interrupt `int 128` to trigger the call

Invoking a shell

Here is the assembly code for a simple system call invoking a shell:

```
        .section .rodata    # data section
args:
        .long arg           # char *["/bin/sh"]
        .long 0             #
arg:
        .string "/bin/sh"

        .text
        .globl  main
main:
        movl    $arg, %ebx
        movl    $args, %ecx
        movl    $0, %edx
        movl    $0xb, %eax
        int     $0x80       # execve("/bin/sh",["/bin/sh"],NULL)
        ret
```

From assembly to shellcode

However, this is not yet quite shellcode: it contains hard-wired (absolute) addresses and a data section.

Question. How could you turn this into position independent code without separate data?

From assembly to shellcode

Moreover, we need to find the binary representation of the instructions (i.e., the compiled shell code).

This will be the *data* that we can then feed back into our attack.

```
$ gcc shellcode.s -o shellcode.out
$ objdump -d shellcode.out
...
080483ed <main>:
 80483ed:   bb a8 84 04 08          mov    $0x80484a8,%ebx
 80483f2:   b9 a0 84 04 08          mov    $0x80484a0,%ecx
 80483f7:   ba 00 00 00 00          mov    $0x0,%edx
 80483fc:   b8 0b 00 00 00          mov    $0xb,%eax
 8048401:   cd 80                   int    $0x80
 8048403:   c3                      ret
```

- We take the hex op code sequence `bb a8 84...` etc and encode it as a string (or URL, filename, etc) to feed into the program as malicious input.

There’s a bit of an art to crafting shellcode for different architectures and scenarios. Handily many examples are online. For example, at shell-storm.org/shellcode or www.exploit-db.com/shellcodes.

Outline

Stack variable corruption

Executable code exploits

- Shellcode
- Redirecting execution

Heap overflows

- Specific heap attacks
- General heap attacks

Summary

Arbitrary code exploit

1. write code useful for an attacker
2. **store executable code somewhere in memory**
3. **use stack overflow to direct execution there**

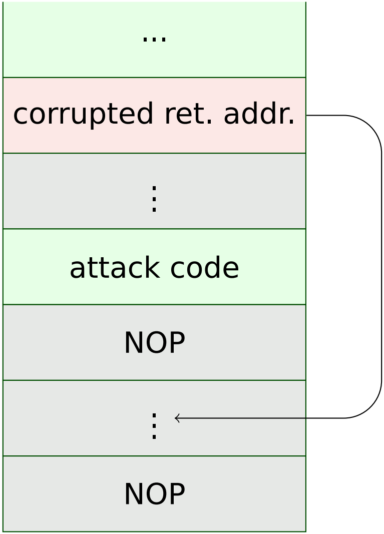
Two options:

- shellcode on stack
- shellcode in another part of the program data

Problem in both cases is :

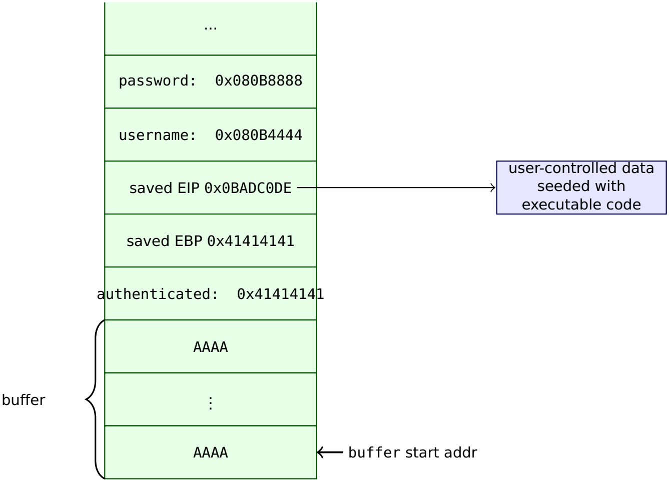
- how to find out where the code is?

Attack code on stack: the NOP sled



The exact address of the attack code in the stack is hard to guess.
The attacker can increase the chance of success by allowing a range of addresses to work.
The overflow uses a *NOP sled*, which the CPU execution "lands on", before being directed to the attack code.

Attack code elsewhere in memory



- Various (sometimes intricate) possibilities
- . . . in an environment variable, modifying function pointers, corrupting caller’s saved frame pointer

Stack smashing without shellcode

Sometimes an attacker cannot directly inject code which gets executed, but can still corrupt return addresses.

Return to library (ret2libc)

The attacker overflows a buffer causing the return instruction to jump to invoke `system()` with an argument pointing to `/bin/sh`.

Return-oriented Programming (ROP)

Sequences of instructions (*gadgets*) from library code are assembled together to manipulate registers, eventually to invoke an library function or even to make a Turing-complete language.

Outline

- Stack variable corruption
- Executable code exploits
  - Shellcode
  - Redirecting execution
- Heap overflows
  - Specific heap attacks
  - General heap attacks
- Summary

Heap overflows: overview

The **heap** is the region of memory that a program uses for dynamically allocated data.

The runtime or operating system provides *memory management* for the heap.

With *explicit* memory management, the programmer uses library functions to allocate and deallocate regions of memory.

Memory safety and undefined behaviour

Memory safety

A programming language enforces *memory safety* if it ensures that reads and writes stay within clearly defined memory areas.

Undefined behaviour

A programming language specification defines the meaning of programs. Without memory safety, the specification may say *the meaning of an illegal memory access is undefined.*

Question. What is the benefit of using "undefined" behaviour in a language spec?

Question. What risks do you see for software security with "undefined" behaviour?

Memory allocation in C

`malloc(size)` tries to allocate a space of `size` bytes.

- It returns a pointer to the allocated region
- . . . of type **`void*`** which the programmer can *cast* to the desired pointer type
- or it **fails** and returns a NULL pointer
- The memory is **uninitialised** so should be written to before being read from

Question. Which points above contribute to (memory) *unsafe* behaviour in C?

Memory allocation in C

`calloc(size)` behaves like `malloc(size)` but it also initialises the memory, clearing it to zeroes.

Question. Suppose we allocate a string buffer, and immediately assign the empty string to it.

What security reason may there be to prefer `calloc()` over `malloc()`?

Memory allocation in C

free(ptr) frees the previously allocated space at ptr.

- No return value (void)
- If it fails (ptr a non-allocated value), what happens?
  - if ptr is NULL, nothing
  - “undefined” otherwise,
  - program may abort, or might carry on and let bad things happen
- What happens if ptr is dereferenced after being freed?
  - depends on behaviour of allocator

Question. Suppose we accidently call free(ptr) before the final dereference of ptr() but before another call to malloc(). Is that safe?

Outline

- Stack variable corruption
- Executable code exploits
  - Shellcode
  - Redirecting execution
- Heap overflows
  - Specific heap attacks
  - General heap attacks
- Summary

Simple heap variable attack

Without memory safety, heap-allocated variables may overflow from one to another.

```c
char *user = (char *)malloc(sizeof(char)*8);
char *adminuser = (char *)malloc(sizeof(char)*8);

strcpy(adminuser, "root");

if (argc > 1)
  strcpy(user, argv[1]);
else
  strcpy(user, "guest");

/* Now we'll do ordinary operations as "user" and
   create sensitive system files as "adminuser" */
```

- Is it possible to overflow user and change adminuser ?

Simple heap variable attack

Problem: how do we know where the allocations will be made?

- Heap allocator is free to allocate anywhere, not necessarily in adjacent memory

Let’s investigate what happens on Linux x86, glibc.

(for a particular version, on a particular day, . . .)

Simple heap variable attack

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

void main(int argc, char *argv[]) {

  char *user = (char *)malloc(sizeof(char)*8);
  char *adminuser = (char *)malloc(sizeof(char)*8);

  strcpy(adminuser, "root");

  if (argc > 1)
    strcpy(user, argv[1]);
  else
    strcpy(user, "guest");

  printf("User is at %p, contains: %s\n", user, user);
  printf("Admin user is at %p, contains: %s\n", adminuser, adminuser);
}
```

```
$ gcc useradminuser.c -o useradminuser.out
$ ./useradminuser.out
User is at 0x9504008, contains: guest
Admin user is at 0x9504018, contains: root
```

```
$ ./useradminuser.out
User is at 0x9483008, contains: guest
Admin user is at 0x9483018, contains: root
```

```
$ ./useradminuser.out frank
User is at 0x8654008, contains: frank
Admin user is at 0x8654018, contains: root
```

- Buffers not adjacent, there’s some extra space
- Addresses not identical each run
- But admin user *is* stored higher in memory!

Let’s try overflowing….

```
$ ./useradminuser.out frank........david
User is at 0x9405008, contains: frank........david
Admin user is at 0x9405018, contains: id
```

Let’s try overflowing. . . .

```
$ ./useradminuser.out frank........david
User is at 0x9405008, contains: frank........david
Admin user is at 0x9405018, contains: id
```

Count more carefully:

```
$ ./useradminuser.out frank56789ABCDEFdavid
User is at 0x9f0b008, contains: frank56789ABCDEFdavid
Admin user is at 0x9f0b018, contains: david
```

Whoa!

Let’s try overflowing. . . .

```
$ ./useradminuser.out frank........david
User is at 0x9405008, contains: frank........david
Admin user is at 0x9405018, contains: id
```

Count more carefully:

```
$ ./useradminuser.out frank56789ABCDEFdavid
User is at 0x9f0b008, contains: frank56789ABCDEFdavid
Admin user is at 0x9f0b018, contains: david
```

Whoa!

Question. Can you think of a way to prevent this attack?

Remarks about heap variable attack

- same kind of attack is possible for (mutable) **global variables**, which are allocated statically in another memory segment
- this is an **application-specific** attack, need to find security-critical path near overflowed variable
- need to be lucky: overwriting intervening memory might cause crashes later, before the program gets to use the intentionally corrupted data

Is there a more generic attack for the heap?

Outline

- Stack variable corruption
- Executable code exploits
  - Shellcode
  - Redirecting execution
- Heap overflows
  - Specific heap attacks
  - General heap attacks
- Summary

Heap allocator implementation

A common heap implementation is to use blocks laid out contiguously in memory, with a *free list* intermingled.

Heap blocks have *headers* which give information such as:

- size of previous block
- size of this block
- flags, e.g., *in-use* flag
- if not in use, pointers to next/previous free block

The doubly-linked free list makes finding spare memory fast for the `malloc()` operation.

Heap allocator implementation

```c
typedef struct mallocblock {
    struct mallocblock *next;
    struct mallocblock *prev;
    int prevsize;
    int thissize;
    int freeflag;
    // malloc space follows the header
} mallocblock_t;
```

- If `freeflag` is non-zero, the block is in the freelist
- Allocator will split blocks and coalesce them again

General heap overflow attack

Rough idea:

- Coalescing blocks unlinks them from the free list
- Attacker makes `unlink()` do an arbitrary write!
  - uses overflow to set next and previous
  - and set flags to indicate free
  - `unlink()` then performs write

Unlinking operation

```c
void unlink(mallocblock_t *element) {
  mallocblock_t *mynext = element->next;
  mallocblock_t *myprev = element->prev;

  mynext->prev = myprev;
  myprev->next = mynext;
}
```

- performs two (related) word writes
  - `mynext->prev=*mynext+2`, `myprev->next=*myprev`
- attacker arranges at least one of these to be useful

Exercise. Check you understand this: draw a picture of a doubly linked list and explain how the attacker can make an arbitrary write.

Writing to arbitrary locations

What locations might the attacker choose?

- *Global Offset Table* (GOT) used to link ELF-format binaries. Allows arbitrary locations to be called instead of a library call.
- *Exit handlers* used in Unix for return from `main()`.
- Lock pointers or exception handlers stored in the Windows *Process Environment Block* (PEB)
- Application-level function pointers (e.g. C++ virtual member tables).

The details are intricate, but library exploits and tookits are available (e.g., Metasploit).

Heap spraying and browser exploits

Apart from operating system (C code) memory management, other application runtimes provide memory allocation features, which may be accessible to an attacker.

A particular case is in **browser-based exploits** which have made use of heaps for managed runtimes such as **JavaScript**, **VBScript**, **Flash**, **HTML5**.

Writing shell code to predictable heap locations is sometimes called **heap spraying**. This is simple in concept: string variables manipulated in scripts are allocated in a heap.

Outline

- Stack variable corruption
- Executable code exploits
  - Shellcode
  - Redirecting execution
- Heap overflows
  - Specific heap attacks
  - General heap attacks
- Summary

Review questions

Stack overflows

- Explain how uncontrolled memory writing can let an attacker corrupt the value of local variables.
- Explain how an attacker can exploit a stack overflow to execute arbitrary code.
- Draw an stack during a stack overflow attack with a NOP sled and shell code, giving addresses

Heap overflows

- Describe the API functions used to interface to heap allocation in C. Give two examples of risky behaviour.
- Show how overflowing one heap-allocated variable can corrupt a second.
- Explain how a heap overflow attack can exploit memory allocation routines to allow arbitrary writes.

Coming next

We’ll look at other kinds of overflow attacks, and some general protection mechanisms.

The best way to understand these attacks is to try them out!

We recommend trying the SEED Labs buffer overflow labs for some good walk-throughs.

References and credits

This lecture included examples from:

- M. Dowd, J. McDonald and J. Schuh. *The Art of Software Security Assessment*, Addison-Wesley 2007.