CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 23rd, 2011

Low level programming bugs

C-programming bugs

There are a few kinds of programming bugs that can lead to serious security exploits:

- Buffer overflows
  - Traditional stack code injection
  - return-to-libc attacks (return oriented programming)
  - Heap overflow attacks
- Double frees
- Integer overflows
- Format string bugs

Stack overflow

The following piece of code illustrates a stack overflow vulnerability:

```
void getuser(int sock)
{
    char buf[1024];
    read(sock, buf, 2048);  /* reading 2048 bytes into 1024 byte buffer is bad */

    /* ... */
}
```

Refresher on how the stack works

Remember that the **stack grows downward**, which means arguments are pushed on the stack starting at higher addresses and going down to lower addresses.

Stack	
0xFFFF	sock
0xFFFB	retaddr
0xFAE9	buf

When the `getuser` function executes:

- Pushes `sock` on the stack
- Pushes the return address `retaddr` on the stack
- Pushes the 1024 byte buffer `buf` on the stack
  - Now, if the address of `retaddr` was x, then the address of the last byte of `buf` has to be $x - 1$, which means the address of the first byte of `buf` will be $x - 512$.
  - This is crucial to understand. The stack grows downward, but the buffer `buf` will grow upward (within the stack). It has to, because `buf`[511] has to be at a higher address than `buf`[0]. That's just how arrays work in C.
  - Also, this is crucial in making the buffer overflow work. Why? Well, if you write `buf`[0], `buf`[1], `buf`[2], …, `buf`[511], and then "accidentally" write `buf`[512] then you wrote over the first byte of `retaddr`.
  - However, even if `buf` grew downward along with the stack, the attack would still be possible, as it will be later explained.

When the function exits, it needs to clean up the arguments and local variables pushed on the stack, so it will pop stuff off the stack by incrementing the stack pointer. Finally, it will jump to the return address which it popped off the stack.

Again: Note that `buf` [0] starts at the bottom and `buf` [1023] is at the top, so if you were to write to `buf` [1024] to `buf` [1027] you would overwrite the 4 bytes of the return address `retaddr`.

Stack	
0xFFFF	`sock`
0xFFFB	`retaddr`
0xFFFA	`buf[1023]`
`buf[i], 0 < i < 1023`	
0xEFD7	`buf[0]`

How can this facilitate an attack?

An attacker can figure out where the buffer `buf` will be in memory by running your program with a debugger and analyzing it.

If your program is vulnerable to a buffer overflow vulnerability, then he can set his username to binary executable code of 1024 bytes length, plus an additional 4 bytes containing the address of the buffer `buf` in memory.

This way when the function `getuser` attempts to return, the program will jump into the buffer `buf` where the attacker code resides and execute this (evil) code.

What's the point of such an attack?

There are a few things an attacker can do if he finds a buffer overflow vulnerability in your program:

- The attacker can **execute arbitrary code** on your machine. He can do bad stuff.
- If your program is privileged, then he can do even more damage and **take over the system**.
- If the attacker can remotely exploit a buffer overflow and your program is running with a lot of privileges, then the attacker could **remotely take over your system**.

Possible fixes for buffer overflows

Fortunately, there are a couple of ways of fixing buffer overflows, some of which can be enforced by the guest OS.

Fix 1 (bad): Buffers should grow downward along with the stack

If buffers were to grow downward along with the stack, such that the address of `buf`[0] is $x + 1023$ and the address of `buf`[1023] is x, then we will show attacks are still possible.

Note that implementing this would be very tedious and abnormal and would interfere with normal pointer semantics: Where should `*(p+1)` take you? 1 byte ahead or 1 byte back?

The reason this would not work as a fix is because there might be an activation record beneath the buffer in the stack, which can be overflown. In our case, the read function was called which pushed some arguments on the stack and also had a return address. As the `read` function fills in our buffer, we can overflow its return address and still succeed with the attack.

Take another look at our code and at the stack layout as the code is executing the `read` function:

```
void getuser(int sock)
{
    char buf[1024];
    read(sock, buf, 2048);  /* reading 2048 bytes into 1024 byte buffer is bad */

    /* ... */
}
```

While the `read` function executes, the stack will be arranged like this:

Stack	
Activation record for `getuser`	
0xFFFF	`sock`
0xFFFB	`retaddr for getuser`
0xFFFA	`buf[0]`
`buf[i], 0 < i < 1023`	
0xEFD7	`buf[1023]`
Activation record for `read`	
0xFED3	`buffer length`
0xEFCF	`buffer pointer`
0xEFCB	`sock`
0xEFC7	`retaddr for read`
other local variables for `read`	

It is now fairly obvious that the attacker can overflow the buffer over `read`'s activation record and change its return address to one of his choice. Therefore, having the buffer grow downward along with the stack will not prevent this attack. In addition, it will also make compiler design more complicated since the address of `buf[0]` is higher than the address of buf[1023] which does not respect normal pointer semantics.

Fix 2 (good): Put buffers on another stack or put return addresses on another stack.

Fix 3 (good): Don't allow executing code on the stack. In general, memory pages should be `W^X` (either writable or executable, but not both).

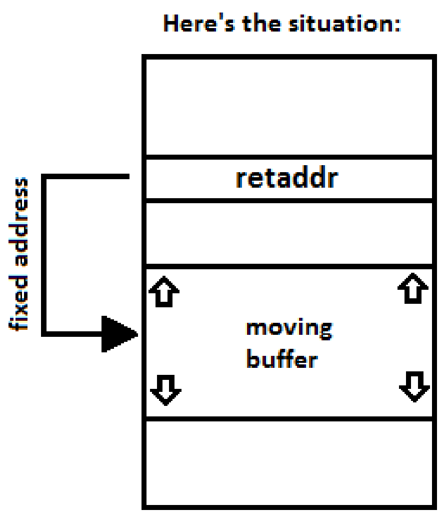
Other good fixes:

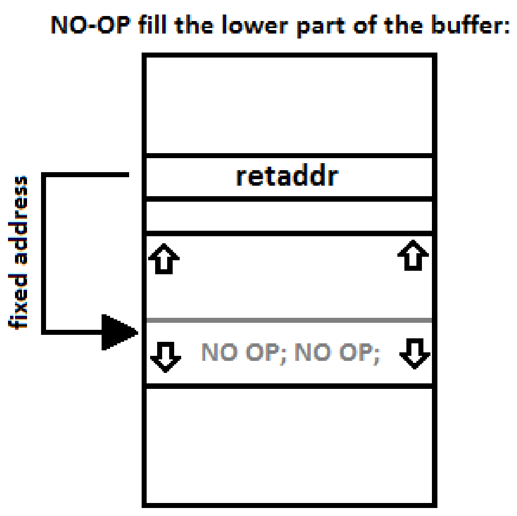
- Address space layout randomization: always pick a random location where the stack will start in virtual memory, rather than having the stack always start at the same fixed memory address which makes the addresses of buffers very predictable
- Canary stack protector: introduce a *random canary variable* between the return address and the local variables of a function. This way, if a local buffer is overflown, the canary will be overwritten. The program can check when it returns whether the canary has remained the same or not and can thus detect a buffer overflow.

Some important details

What about guessing the buffer address? It turns out the buffer address will shift around a little bit. The stack gets shifted around because the command-line arguments and the environment-variables of the program are pushed onto the stack initially when the program starts up.

For the attacker this is not a big issue. The attacker can have a range where the buffer address is likely to be in. To ensure that he can jump to the code in the buffer, he can put "landing zone" code in the beginning of the buffer, which are just no-op instructions such that when the function jumps some of the landing code is executed until the real code starts. If he did not have landing code, then the program would jump into the middle of the buffer executing code there and skipping the previous code, which would not make sense.





Return-to-libc attacks

To get past the W^X stack overflow protection mechanism, **return-to-libc attacks** were invented.

In a return-to-libc attack, the attacker sets the overflown return address to a function in **libc** (the standard C library), such as `system`. This will get past the W^X protection since the libc memory pages are executable.

The `system` libc function takes one argument. The attacker is going overflow `buf` into `retaddr` such that it points to `system`, but he also needs to take care of the stack above `retaddr` such that `system` can make sense of it and grab its argument from it.

`system` will expects the following data on the stack: a pointer to a string representing the command and a return address of its own

The command string pointer can point somewhere inside our overflown buffer. Also, the landing pad for the command string can be just a bunch of semicolons.

Stack before overflow		**Desired stack after overflow**	
Activation record for the caller of `getuser`		**Forged `system` activation record**	
`...`		0x10007	`cmd pointer (4 bytes)`
0x10003	`local variables, retaddr, etc.`	0x10003	`retaddr of system`
Activation record for `getuser`		**Activation record for `getuser`**	
0xFFFF	`sock`	0xFFFF	`sock`
0xFFFB	`retaddr for getuser`	0xFFFB	`retaddr = &system`
0xFFFA	`buf[1023]`	0xFFFA	`buf[1023]`
`buf[i], 0 < i < 1023`			`buf[i], 0 < i < 1023`
0xEFD7	`buf[0]`	0xEFD7	`buf[0]`

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 23rd, 2011

Return-oriented programming

In the "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" paper, Hovav Shacham demonstrates how to mount an attack to execute arbitrary code after a buffer overflow in the absences of libc functions.

Excerpt: "*The building blocks for the traditional return-into-libc attack are functions, and these can be removed by the maintainers of libc. By contrast, the building blocks for our attack are short code sequences, each just two or three instructions long. Some are present in libc as a result of the code-generation choices of the compiler. Others are found in libc despite not having been placed there at all by the compiler. In either case, these code sequences would be very difficult to eliminate without extensive modifications to the compiler and assembler.*"

The main idea: The attacker finds snippets of code that end in a return (jump instruction). He builds a list of such code snippets and their starting addresses. He then overflows the stack with return addresses such that when he makes the first jump to the first snippet, that snippet will return to the next snippet, which will return to the next snippet, and so on. This way, the attacker executes arbitrary code (limited to the kinds of snippets he can find) on the machine.

Non-control-data overflows

In all of the presented attacks we hijacked the control flow of the program. Essentially, we caused the program to go into places it would normally not go to. However, sometimes you can change the flow of a program by modifying its variables. This flow change can be very useful for instance if you can change the flow inside a `is_super_user` function, such that it always returns yes.

```
int is_super_user(char * username)
{
    int result;

    char buf[1024];

    db_lookup(username, userinfo);
    result = userinfo->id < 1024;

    //  logs something
    strcpy(buf, username);

    /* ... */

    return result;
}
```

Attack: If the attacker sends in a long enough user name, he can overflow `buf` and the first thing he will write over will be the `result` variable, which would trick the function to return 1, indicate the user is a super-user.

Intra-struct overflows

An attacker can use buffer overflows to change the values inside a stack or heap-allocated `struct`. Imagine changing the `is_admin` field of the `userinfo` struct from false to true. It could be very useful >:)

```
struct userinfo {
    char username[16];
    int is_admin;
}

struct userinfo * login_user(char * username)
{
    struct userinfo * result = malloc(...);

    result->is_admin = ...;

    strcpy(result->username, username);

    return result;
}
```

Problem: Unclean user input. The attacker can provide a username longer than whatever was malloc'd and he can overflow the `userinfo` struct, changing the `is_admin` field to true.

Fix: Use `strncpy` instead of `strcpy`.
Better, general fix: Only use `snprintf` when dealing with strings. It's clear, clean and nice.

```
int snprintf(char * dest, int size, const char * fmt, ...);
```

`snprintf` **always null terminates** and it **tells you how long your string would be** if you had enough space, so you can use to pre-allocate the exact amount of needed space