Spyrou Michalis, May 2014

BUFFER OVERFLOW ATTACKS AND PROTECTION MECHANISMS (v2)

whoami

- http://stack0verflow.wordpress.com/ My blog (rarely updates)

- @mpekatsoula on twitter

- Interested in HPC, programming and security

What is an overflow

- Simply put, try to fit 2L of water inside a 1,5L bottle.
- Now change L with bytes.
- What do you think will happen?

Quiz

```
void foo(int z, int x) {
        char array[12];
        gets(array);
}
```

Quiz

-
  ```
  void foo(int z, int x) {
          char array[12];
          gets(array);
  }
  ```

- What will happen if input is bigger than 12 bytes?

Quiz

```
void foo(int z, int x) {
        char array[12];
        gets(array);
  }
```

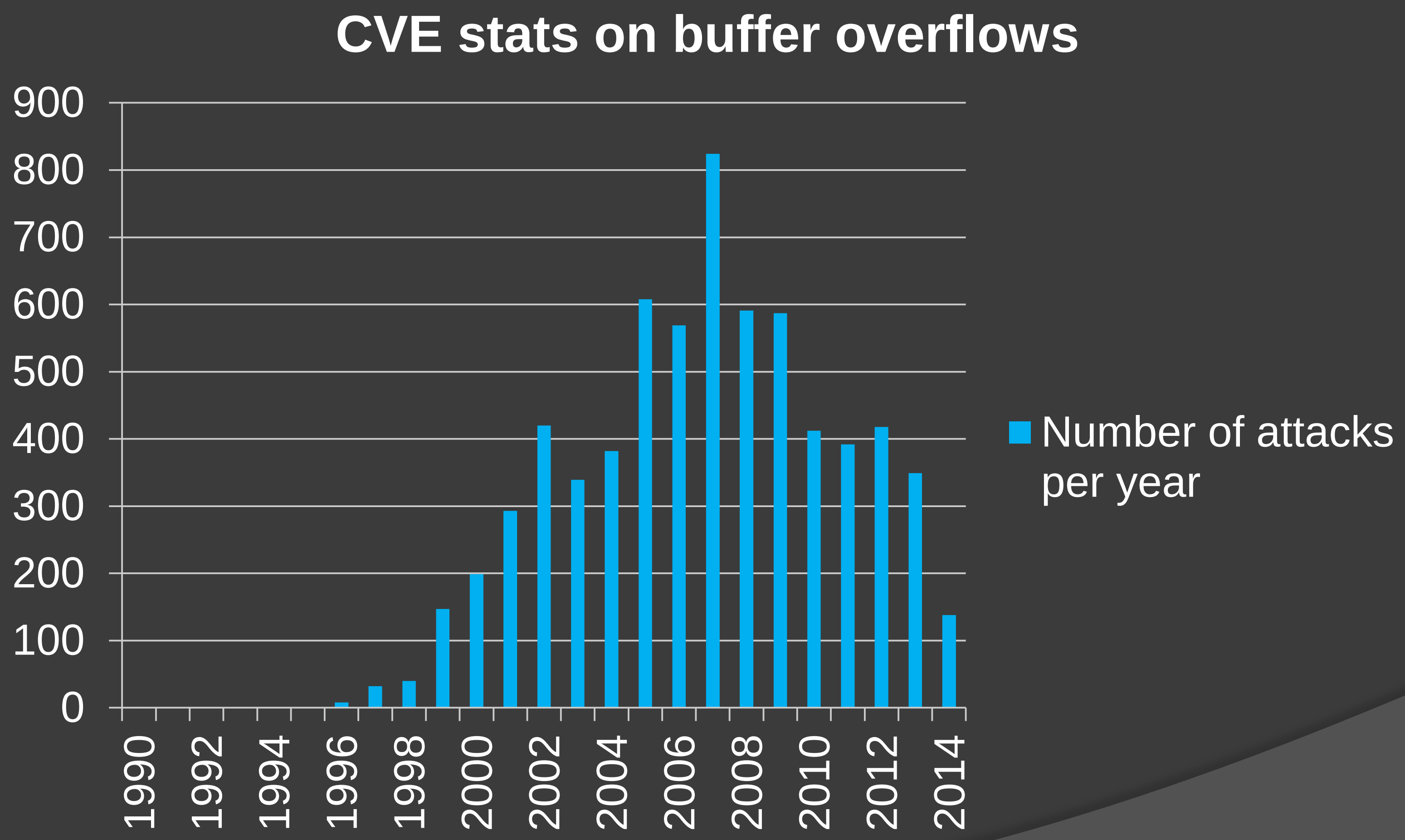
- What will happen if input is bigger than 12 bytes?

- **A)** Segfault **B)** Undefined **C**) The environment/kernel/runtime/compiler is smart, there will be a warning/error **D)** Release the Kraken

Some history

- First publicly documented in 1972 (pdf)
- First malicious exploitation in 1988 by Morris worm
- In 1996, Elias Levy (aka Aleph One) published in Phrack magazine the paper "Smashing the Stack for Fun and Profit"

Attacks statistics

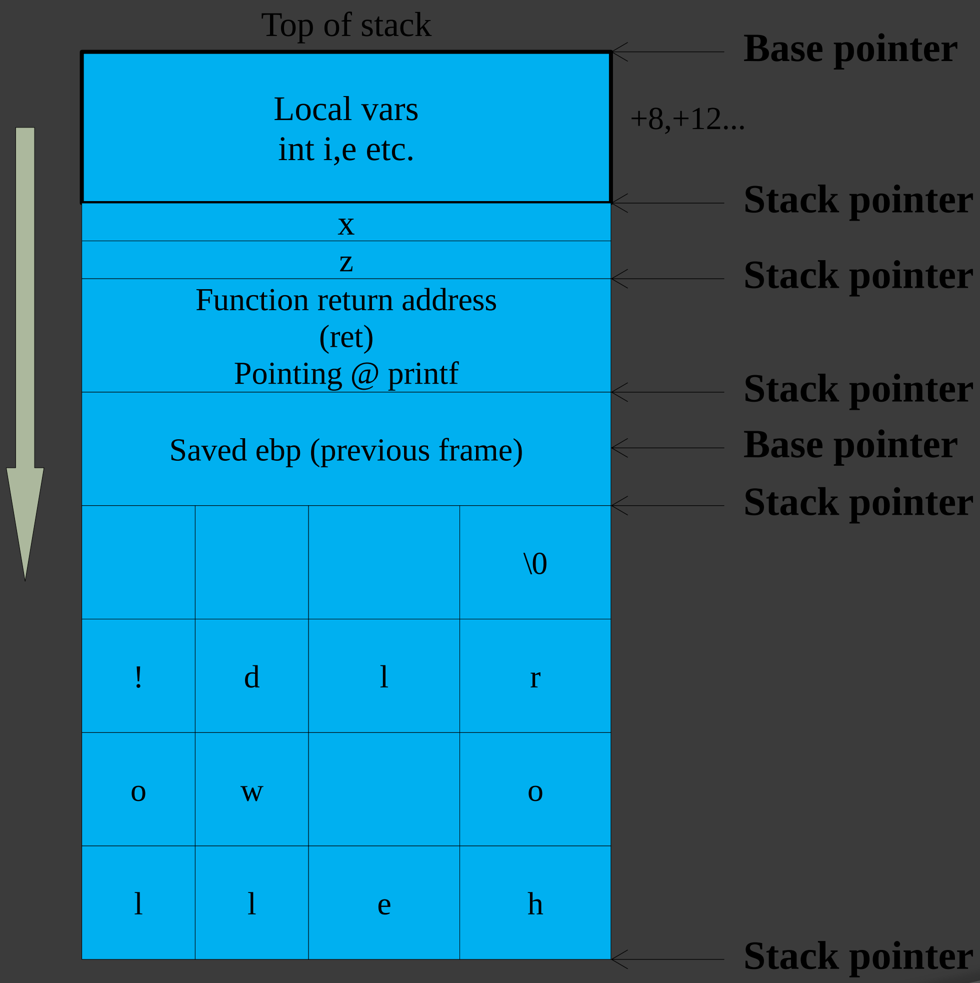


Data taken from: http://web.nvd.nist.gov

Categorization

- **Stack** based
  - Exploits the functionality of the stack
- **Heap** based
  - More difficult to exploit
  - Not in this context

How stack works (1/2)



Top of stack

Local vars
int i,e etc.

+8,+12...

x

z

Function return address
(ret)
Pointing @ printf

Saved ebp (previous frame)

			\0
!	d	l	r
o	w		o
l	l	e	h

Base pointer

Stack pointer

Stack pointer

Stack pointer

Base pointer

Stack pointer

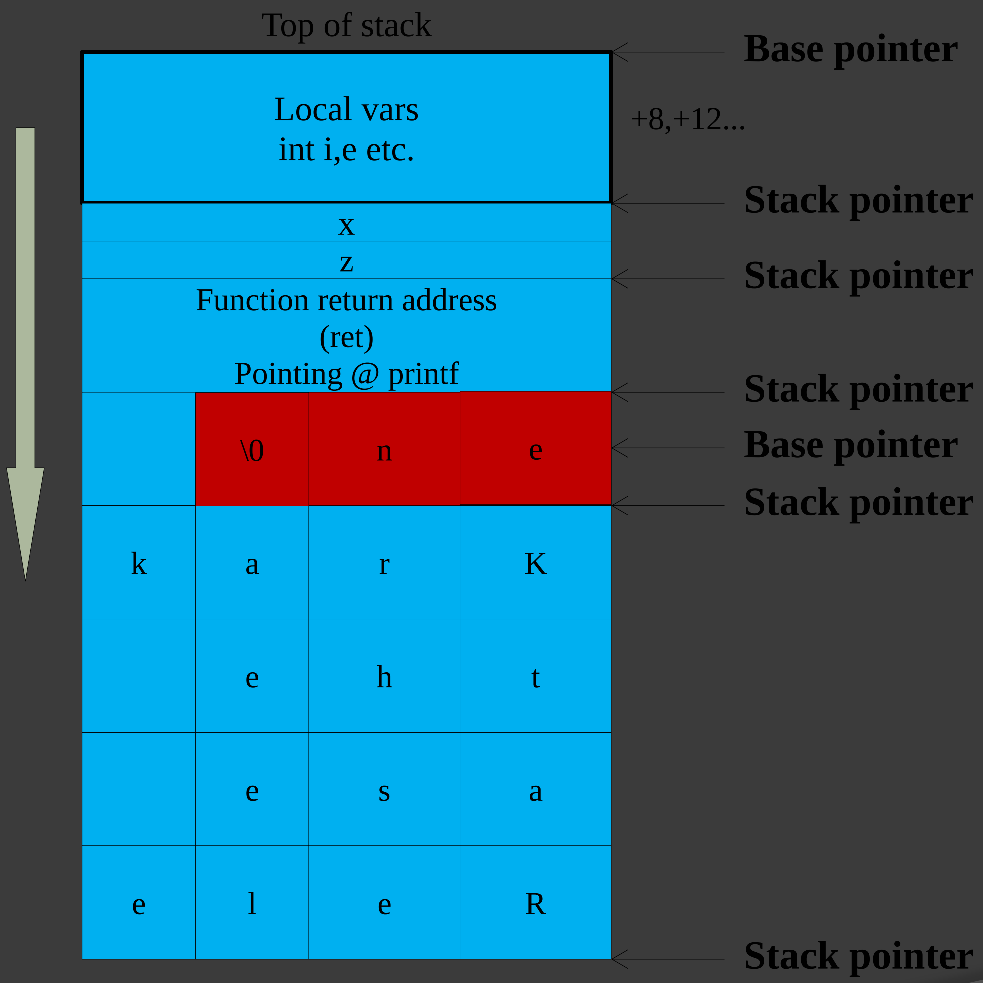
Stack pointer

```
void foo(int z, int x) {
  char array[12];
  gets(array);
}

int main( void ) {
  int i,e;
  foo(i,e);
  printf(“Bye\n”);
}
```

Input: hello world

How stack works (2/2)



Top of stack

Local vars
int i,e etc.

+8,+12...

x

z

Function return address
(ret)
Pointing @ printf

	\0	n	e
k	a	r	K
	e	h	t
	e	s	a
e	l	e	R

Base pointer

Stack pointer

Stack pointer

Stack pointer

Base pointer

Stack pointer

Stack pointer

```
void foo(int z, int x) {
  char array[12];
  gets(array);
}

int main( void ) {
  int i,e;
  foo(i,e);
  printf(“Bye\n”);
}
```

Input: Release the Kraken

So what?

- Well, we can overwrite **$ebp**, not much here..

So what?

- Well, we can overwrite $ebp, not much here..

- What about **ret**?

Shellcoding is an art (1/2)

- **Shellcode** is a small piece of code/instructions injected and executed by an exploited program
- The main idea:
  - Write it in a high level language
  - Extract the opcodes from the assembly executable
  - Must be small and null free

Shellcoding is an art (2/2)

- Step 1

```
#include <stdio.h>
 int main(void){
   char *shell[2];
   shell[0] = "/bin/sh";
   shell[1] = NULL;
   execve(shell[0],shell,NULL);
  }
```

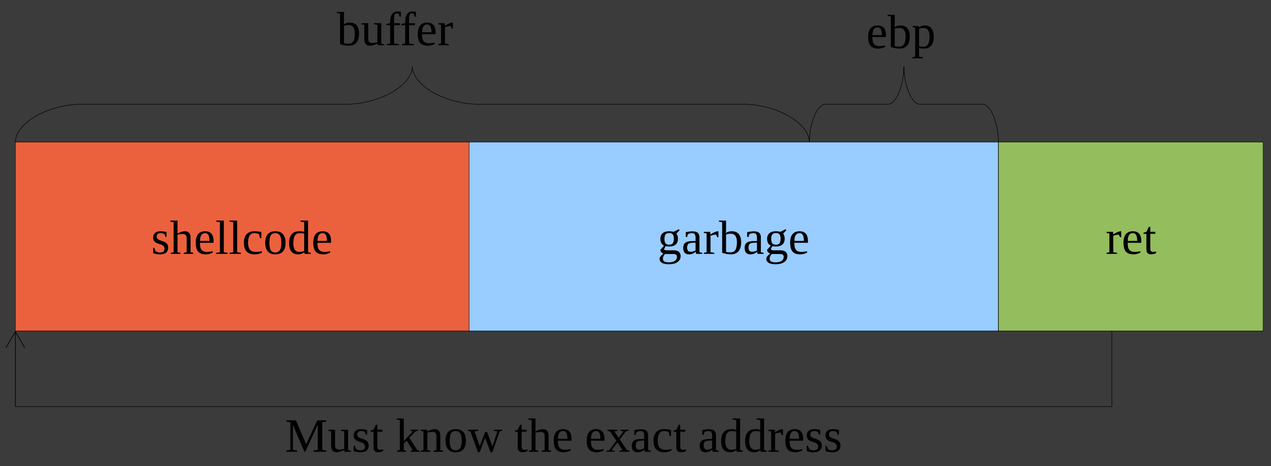
- Step 2

```
0x0804f740 <+0>:   push   %ebp
0x0804f741 <+1>:   mov    %esp,%ebp
0x0804f743 <+3>:   mov    0x10(%ebp),%edx
0x0804f746 <+6>:   push   %ebx
0x0804f747 <+7>:   mov    0xc(%ebp),%ecx
0x0804f74a <+10>: mov    0x8(%ebp),%ebx
0x0804f74d <+13>: mov    $0xb,%eax
0x0804f752 <+18>: int    $0x80
0x0804f754 <+20>: cmp    $0xfffff000,%eax
0x0804f759 <+25>: ja     0x804f75e <execve+30>
0x0804f75b <+27>: pop    %ebx
0x0804f75c <+28>: pop    %ebp
0x0804f75d <+29>: ret
0x0804f75e <+30>: mov    $0xffffffe8,%edx
0x0804f764 <+36>: neg    %eax
0x0804f766 <+38>: mov    %gs:0x0,%ecx
0x0804f76d <+45>: mov    %eax,(%ecx,%edx,1)
0x0804f770 <+48>: or     $0xffffffff,%eax
0x0804f773 <+51>: jmp    0x804f75b <execve+27>
```

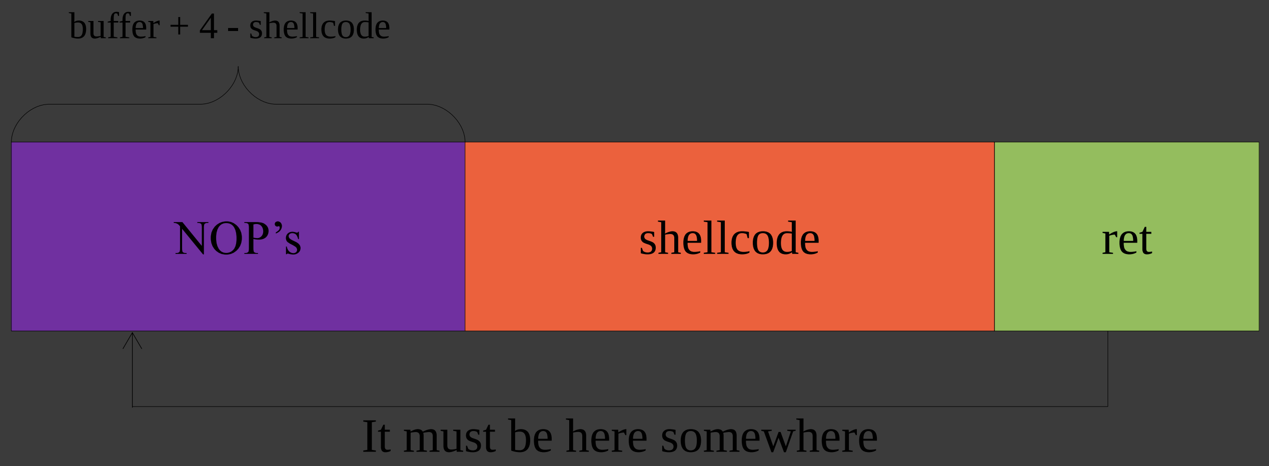
Step 3

```
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"
```

A simple technique



NOP sled


buffer + 4 - shellcode
NOP’s
shellcode
ret
It must be here somewhere

Common exploits

- Privilege Escalation
  - A program is running with root privileges and it's forced to execute arbitrary code
- Worm
  - A program searches for vulnerable systems and exploits them
- Denial of service

Part 2: Protection Mechanisms

A lot have changed

- Compilers got smarter
- Kernel got smarter
- Programmers got smarter (?)
  - Well it’s debatable

Non-Executable stack (nx-stack)

- Code on stack cannot be executed

Non-Executable stack (nx-stack)

- Code on stack cannot be executed

- Problem solved!

Non-Executable stack (nx-stack)

- Code on stack cannot be executed

- Problem solved!
  - Or not?

Non-Executable stack (nx-stack)

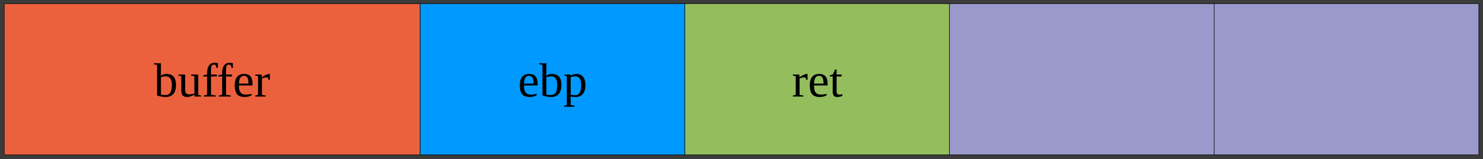
- Code on stack cannot be executed

- Problem solved!
  - Or not?

- The code can be executed somewhere else
  - ret2libc
  - ret2data

ret2libc

int system(const char *command);



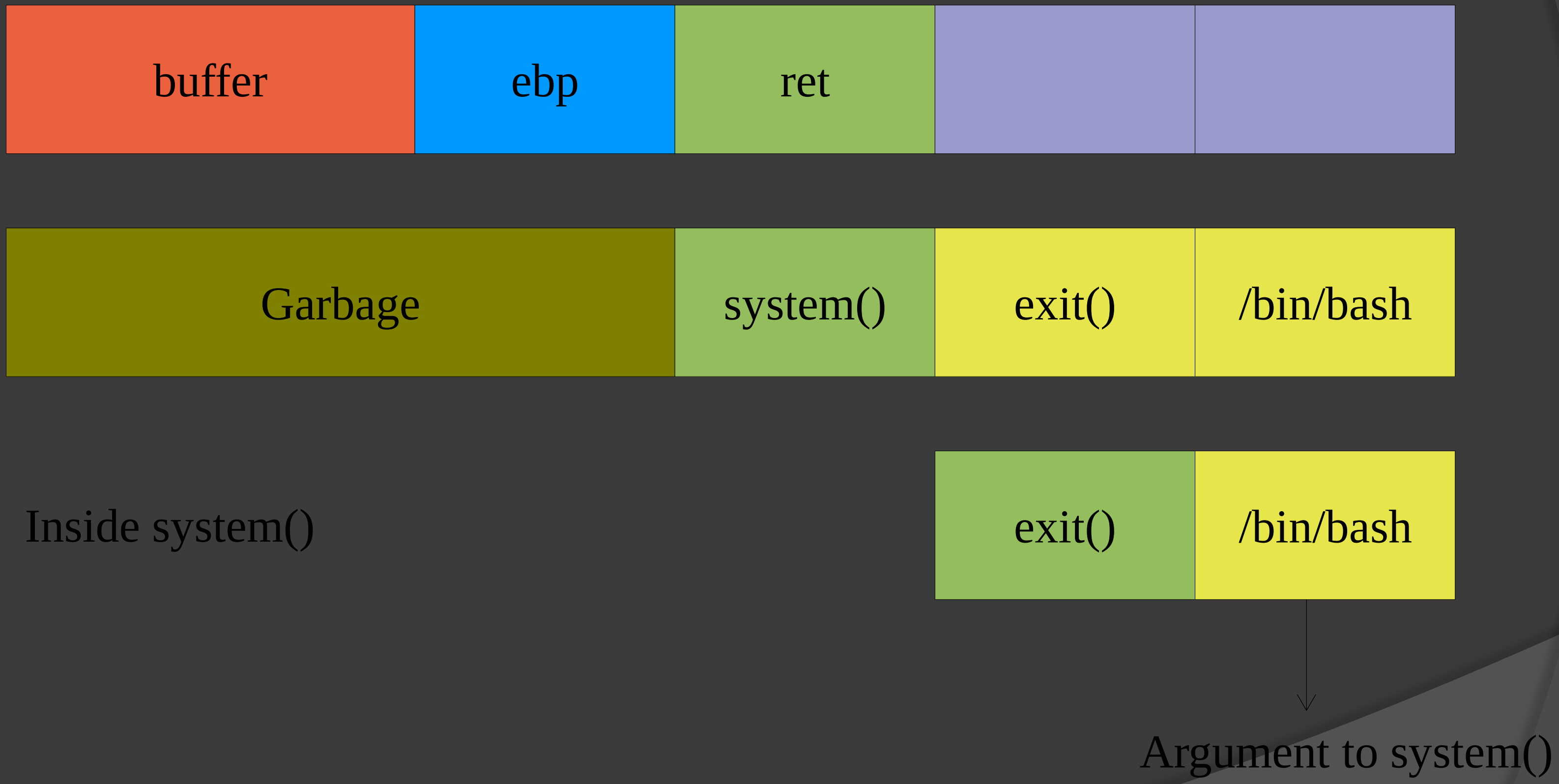
buffer

ebp

ret

ret2libc

int system(const char *command);



buffer | ebp | ret

Garbage | system() | exit() | /bin/bash

Inside system() | exit() | /bin/bash

Argument to system()

W^X - Either Writable or Executable Memory

- First appeared in OpenBSD 3.3, released May 2003

W^X - Either Writable or Executable Memory

- First appeared in OpenBSD 3.3, released May 2003
- Again, not secure enough.

W^X - Either Writable or Executable Memory

- First appeared in OpenBSD 3.3, released May 2003
- Again, not secure enough.
  - May there is already a code in the program that help us do our job

W^X - Either Writable or Executable Memory

- First appeared in OpenBSD 3.3, released May 2003
- Again, not secure enough.
  - May there is already a code in the program that help us do our job
  - Also a portion of the memory might be W+X
    - ret2strcpy

W^X - Either Writable or Executable Memory

- First appeared in OpenBSD 3.3, released May 2003
- Again, not secure enough.
  - May there is already a code in the program that help us do our job
  - Also a portion of the memory might be W+X
    - ret2strcpy
  - Change a memory region from W^X to W+X
    - mprotect()

Address Space Layout Randomization

- Linux has enabled a weak form of ASLR by default since kernel version 2.6.12 (released June 2005).

Address Space Layout Randomization

- Linux has enabled a weak form of ASLR by default since kernel version 2.6.12 (released June 2005).

- The address of: stack, heap, libraries & executable application are randomized

Address Space Layout Randomization

- Linux has enabled a weak form of ASLR by default since kernel version 2.6.12 (released June 2005).

- The address of: stack, heap, libraries & executable application are randomized

- Still:
  - Brute force: on 32-bit arch, only 24bits
  - Some areas may not be randomized

Canaries (1/3)

- Compiler mechanism

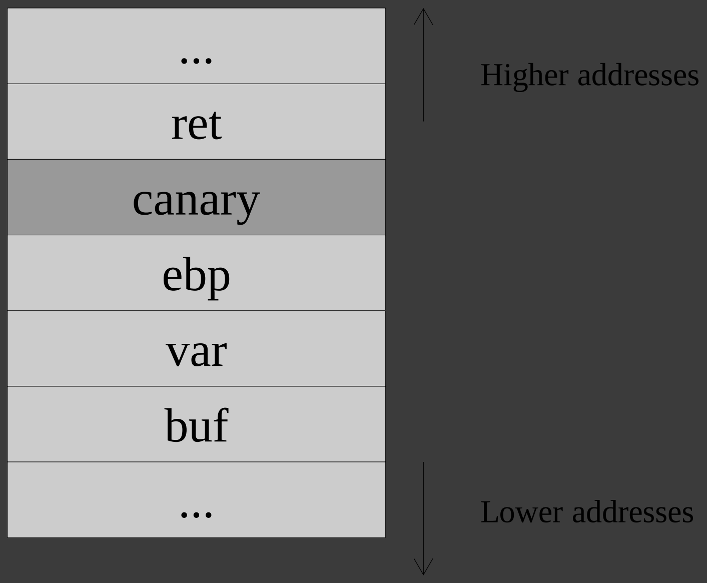
Canaries (1/3)

- Compiler mechanism
- Placed between buffers and sensitive information

Canaries (1/3)

- Compiler mechanism
- Placed between buffers and sensitive information
- Common types:
  - Terminator: 0x000aff0d
  - Random: 0x0823beef
  - NULL: 0x00000000

Canaries (2/3)



...

ret

canary

ebp

var

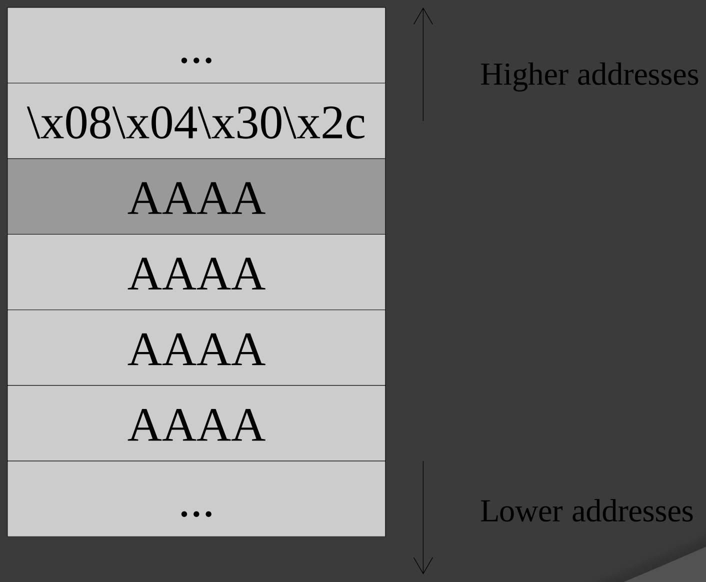
buf

...

Higher addresses

Lower addresses

- If we try to overwrite ret, canary will be overwriten too.



...

\x08\x04\x30\x2c

AAAA

AAAA

AAAA

AAAA

...

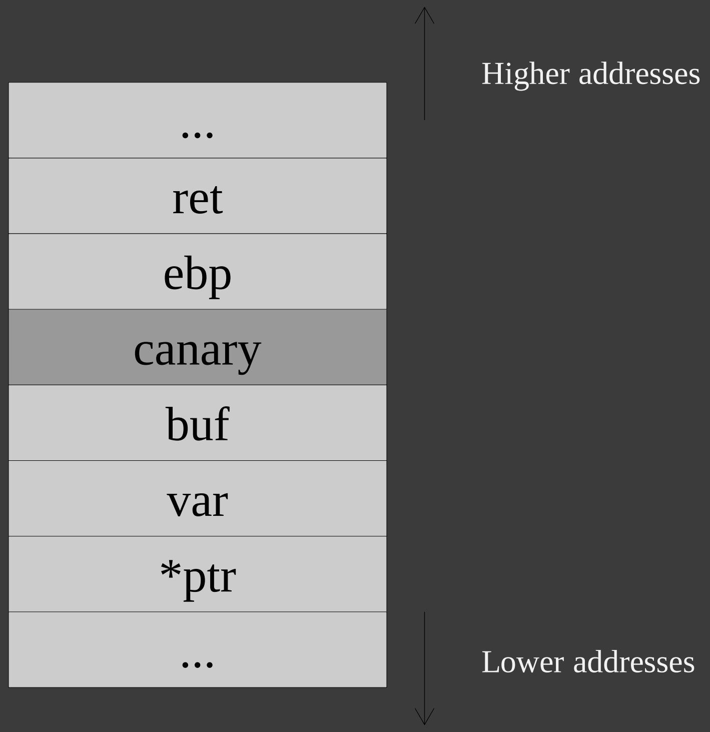
Higher addresses

Lower addresses

- The overflow will be detected and the program will be killed

Canaries (3/3)

- Possible to bypass



Higher addresses

...

ret

ebp

canary

buf

var

*ptr

...

Lower addresses

- Possible brute force (fork())
- In reallity the things are much more complicated
- For example gcc tryies to create a safe frame, by reordering data

Ascii Armored Address Space

- Load all shared libraries in addresses beginning with a null byte: 0x00110000
- Not so secure on little-endian platforms

More techniques

- Bounds checking
  - Compiler adds runtime checks for each allocated bock of memory (C/C++)

Are BO’s still a threat?

- Definitely. Although remote attack is nearly impossible

Are BO’s still a threat?

- Definitely. Although remote attack is nearly impossible

- On a default Ubuntu installation:
  - gcc with SSP
  - glibc with heap protection
  - ASLR
    - stack, libs, exec, brk..

Why all this

- Surely all this is because newbies write bad code.

Why all this

- Surely all this is because newbies write bad code.
- Actual code on openjpeg:

```
case 'f' : /* floats */ {
        char tmp[16];
        double value = va_arg(arg, double);
        sprintf(tmp, "%f", value);
        strcat(message, tmp);
        j += strlen(tmp); ++i; break;
}
```

Even Apple did wrong

- Feb 2014 Apple published their Secure Coding Guide.

Even Apple did wrong

- Feb 2014 Apple published their Secure Coding Guide.

- Code

  -
    ```
    size_t bytes = n * m; // signed integers.
    if (n > 0 && m > 0 && SIZE_MAX/n >= m) {
        … /* allocate “bytes” space */
    }
    ```

Conclusion

- Security is hard. Writing secure code is harder
- Always remember to check boundaries
- Use strncat instead of strcat etc.

I want more security

- You can patch your kernel with grsecurity set of patches (http://grsecurity.net/)

I want more security

- You can patch your kernel with grsecurity set of patches (http://grsecurity.net/)
- One of them is PaX

PaX

- Restricted mprotect()
- ASLR
- Enforced non-executable pages (use of NX bit or emulate the NX bit)
- Does not allow a page to be W+X or X after W
- And much more..

Playtime

- You can try Hardened Gentoo (http://www.gentoo.org/proj/en/hardened/) a security-enhanced version of Gentoo Linux
- If you want to test your skills: http://community.corest.com/~gera/InsecureProgramming/

EOF

Thanks for watching!

Questions?