Information Security
CS3002
(Sections BDS-7A/B)
Lecture 15

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science

09 October, 2024

Malware analysis

- Malware analysis is the study or process of determining the functionality, origin and potential impact of a given malware. (Wikipedia)
- Three typical use cases:
  - Computer Security incident management
  - Malware research
  - Indicators Of Compromise (IOCs) extraction
    - See Malware Analysis: Steps & Examples – CrowdStrike
- Types:
  - Static
  - Dynamic

Why analyze malware?

- To assess damage
- To discover *indicators of compromise* (IOCs)
- To determine sophistication level of an intruder
- To identify a vulnerability
- To catch the "*bad guy*"
- To answer business related questions
  - How long has it been here, spreads on its own, etc.?
- To answer technical questions
  - Date of installation, compilation, persistence mechanism, network or host based indicators

Static Analysis

- Analysis in which code is not executed
- “Dead” code is read and understood
- Also referred to as: code analysis
- Requires peeking into the code using a hex editor, unpacking and performing string searches
- Disassembling the malware. Disassemblers take machine code to higher-level code
  - IDA Pro
- Static analysis is safer
- Malware files are fingerprinted before analysis. Just in case malware analysis is being expected by the (malware) developer
- Virus scan:
  - PEiD, Caprica6 tool can tell you about “packed” code

Dynamic Analysis

- Conducted by observing and manipulating malware as it runs
- Needs a safe environment to analyze (run) the code
  - Sandboxed environment
- Requires monitoring the system
  - Registry files activity
  - File and process/system level activity
  - Network level activity
- Some tools:
  - Wireshark
  - SysInternals process monitor
  - netstat or ResMon in Windows can be used
- Requires analysis while the code is being run using tools like WinDbg

Static vs. Dynamic Analysis

- Static: Dissecting code via different resources without executing
- Dynamic: Behavioral analysis is performed by executing the malware.
- Static is much slower (and exhaustive at times) as compared to dynamic.
- Static is far safer than dynamic.
- Static doesn’t (necessarily) need a sandboxed environment while dynamic does.

Six Steps to incident handling process

- ***Preparation***: Get our team ready. Jump bags, warning banners, response strategies
- ***Identification***: Identify if an event is an incident. Done at network perimeter level or host/system level.
- ***Containment***: limit the propagation/spreading of malware incident.
- ***Eradication***: Removal of infection from the system.
- ***Recovery***: Restoration of services/functionalities
- ***Lessons learned***: Be prepared for next time. Study the reason why an incident occurred and take care of it so it wont get repeated.

Malware defenses (1)

- ***Detection***: once the infection has occurred, determine that it has occurred and locate the malware
- ***Identification***: once detection has been achieved, identify the specific virus that has infected a program
- ***Removal***: once the specific malware has been identified, remove the malware from the infected program and restore it to its original state

Malware defenses (2)

- The *first generation scanner*
  - Malware signature (bit pattern)
  - Maintains a record of the length of programs
- The *second generation scanner*
  - Looks for fragments of code (neglect unnecessary code)
  - Checksum of files (integrity checking)
- The *third generation scanner*
  - Identify a malware by its actions
- The *fourth generation scanner*
  - Include a variety of anti-malware techniques

Malware defenses (3)

- Malware-specific detection algorithm
  - Deciphering
  - Filtering
- Collection method
  - Using honeypots
- Analyze program behavior
  - Network access
  - File open
  - Attempt to delete file
  - Attempt to modify the boot sector

How to prevent them?

- Simple! Learn about security (Not so simple)
- Use a secure Operating systems
- Use secure browsers and plugins/extensions
- And update/patch regularly
- Install anti-virus (maybe?)
- Avoid torrents
- Surf secure websites
- Don’t download what you don’t understand/need
- Use Instant Messaging apps carefully
- Keep backups

How to prevent them?

- Don’t install software that you don’t need or remove after one time use(worms!).
- Install software carefully. Unnecessary bundles gets installed
- Open email attachments with caution
- Monitor the performance of your pc regularly
- Keep frequent restore points and restore your pc if you think you executed a virus/worm/trojan
- Avoid unlicensed software installation
- Layers of authorization for installation of new tools/software

How to prevent them? – Two Layers

- Personal vigilance (first layer)
  - Knowing what to do and what to install
  - Understanding of the system and security
  - Strong passwords (password checkers)
- Protective tools (second layer)
  - Effective and enough prevention tools
  - They are never enough

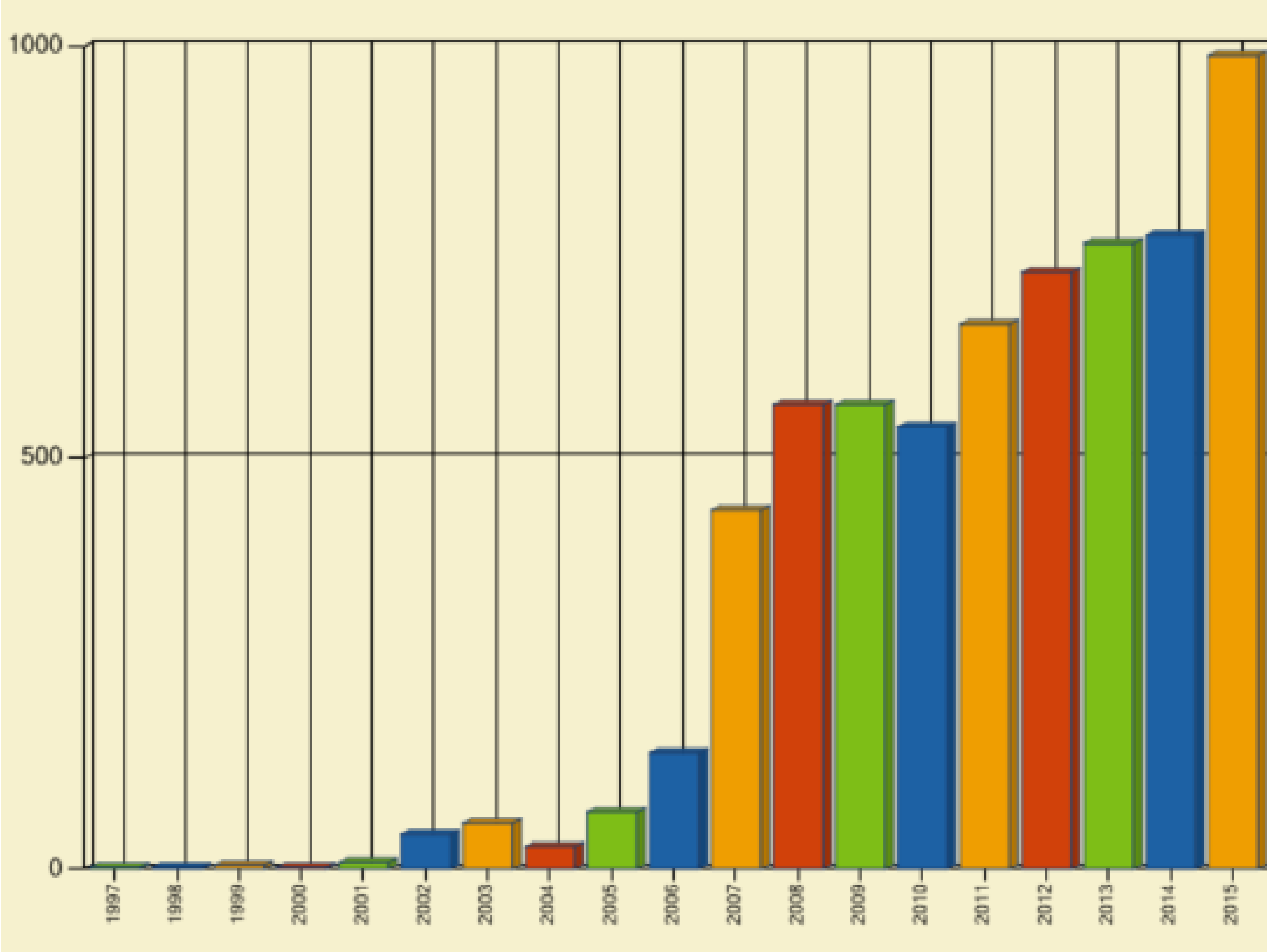
Software Security: Control Hijacking Attacks

- Attacker's goal:
  - Take over target machine (e.g., web server)
    - Execute arbitrary code on target by hijacking application control flow
- Examples:
  - Buffer overflow and integer overflow attacks
  - Format string vulnerabilities
  - Use after free

First Example: Buffer Overflows

- Extremely common bug in C/C++ programs
    - First major exploit: 1988 Internet worm → `fingerd`

Source: web.nvd.nist.gov →

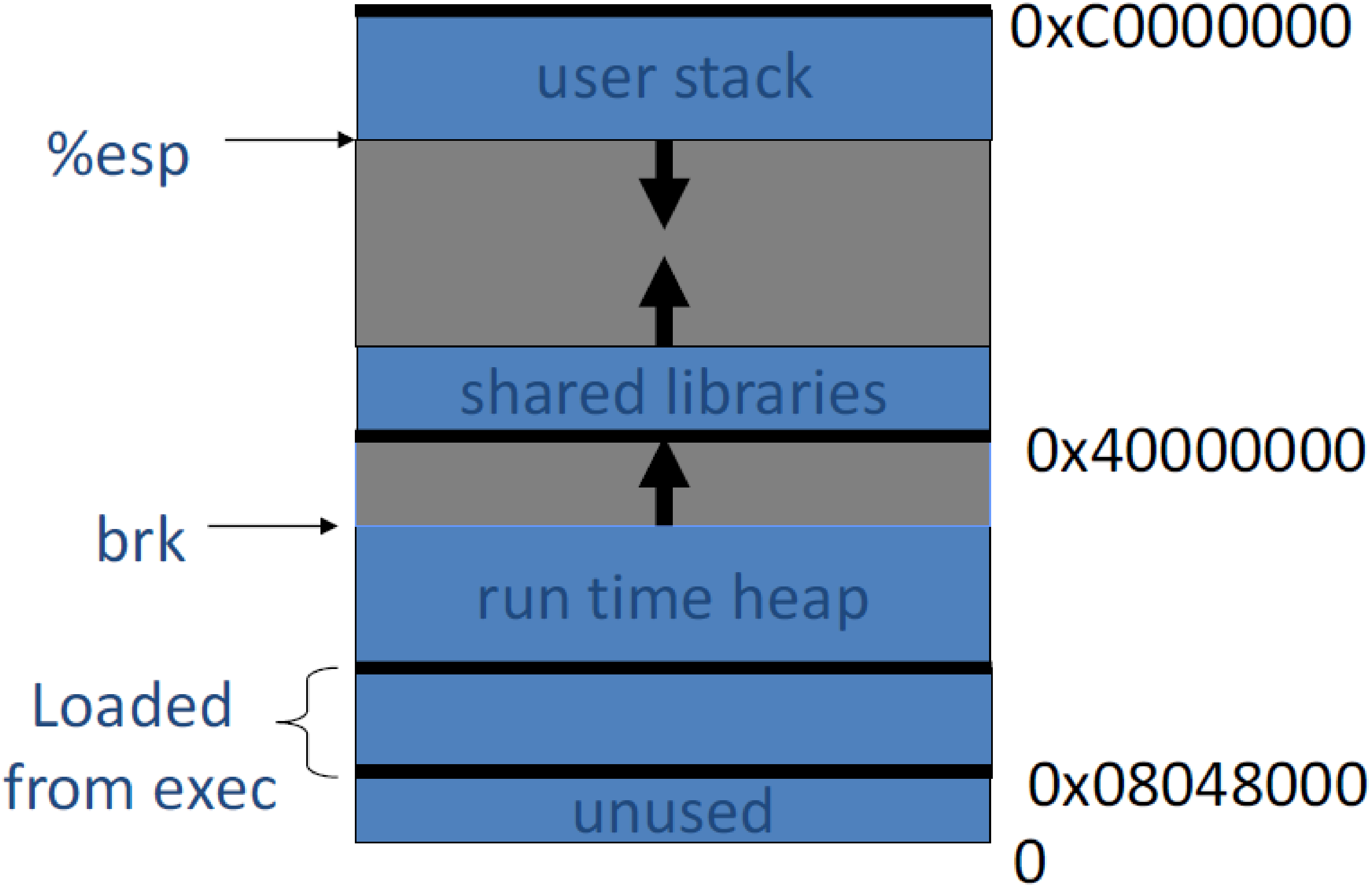


What is needed?

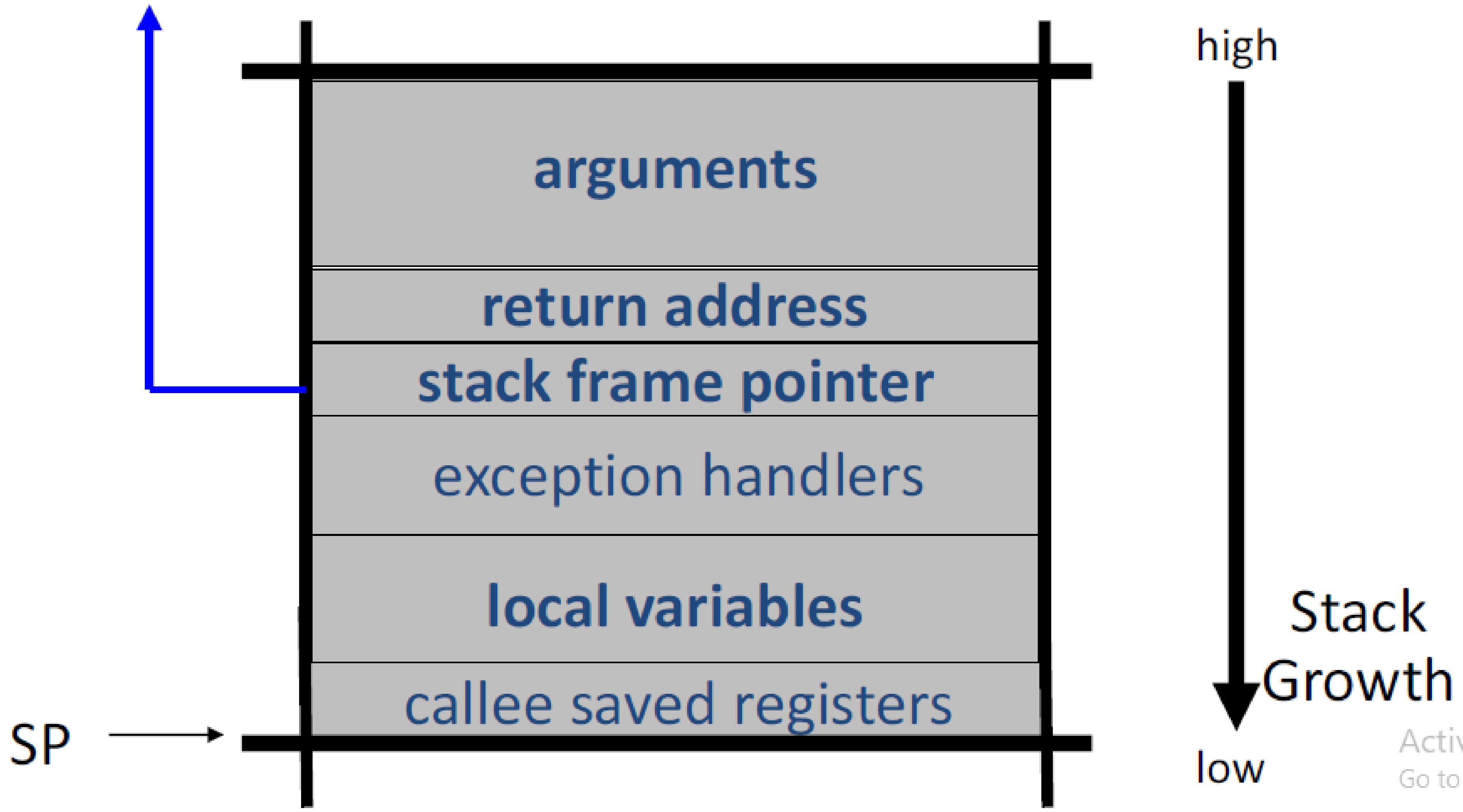
- Understanding C functions, the stack, and the heap.
- Know how system calls are made
- The exec() system call

- Attacker needs to know which CPU and OS used on the target machine:
  - Our examples are for x86 running Linux or Windows
  - Details vary slightly between CPUs and Operating Systems:
    - *Little endian versus big endian (x86 vs, Motorola)*
    - *Stack Frame structure (UNIX vs. Windows)*

Linux process memory layout



Stack Frame



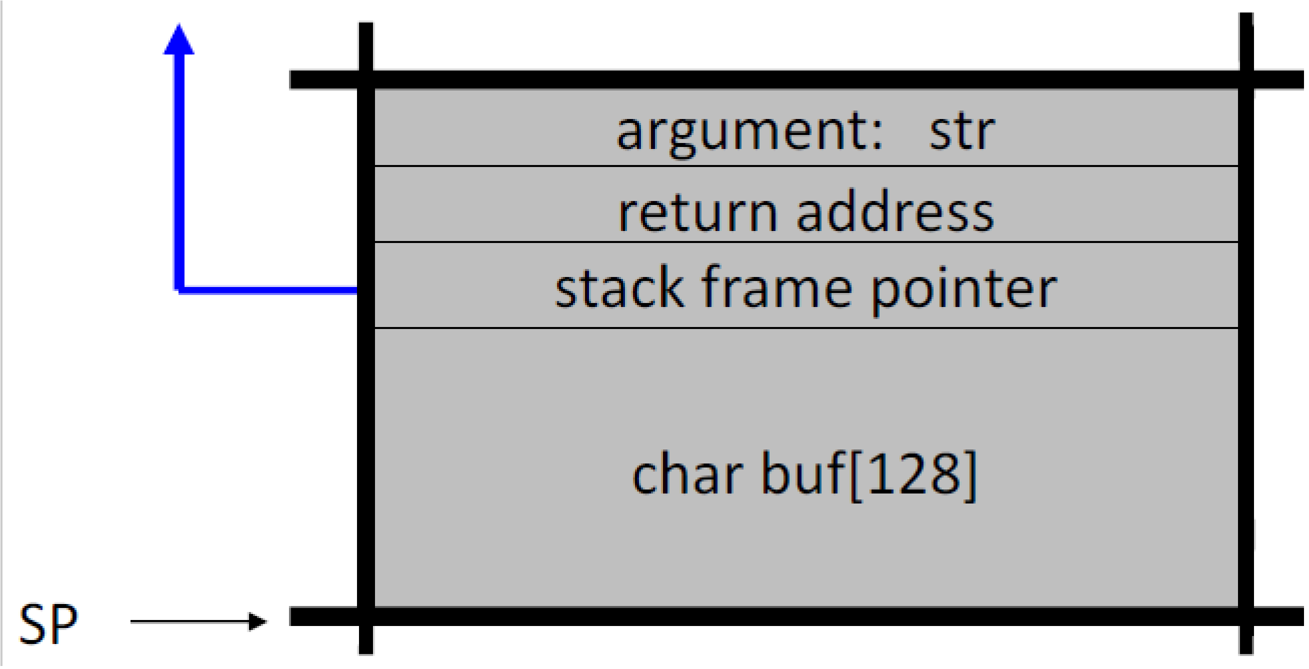
arguments
return address
stack frame pointer
exception handlers
local variables
callee saved registers
SP
high
Stack Growth
low

What are buffer overflows?

Suppose a web server contains a function:

```
void func(char *str)
{
   char buf[128];
   strcpy(buf, str);
   do-something(buf);
}
```

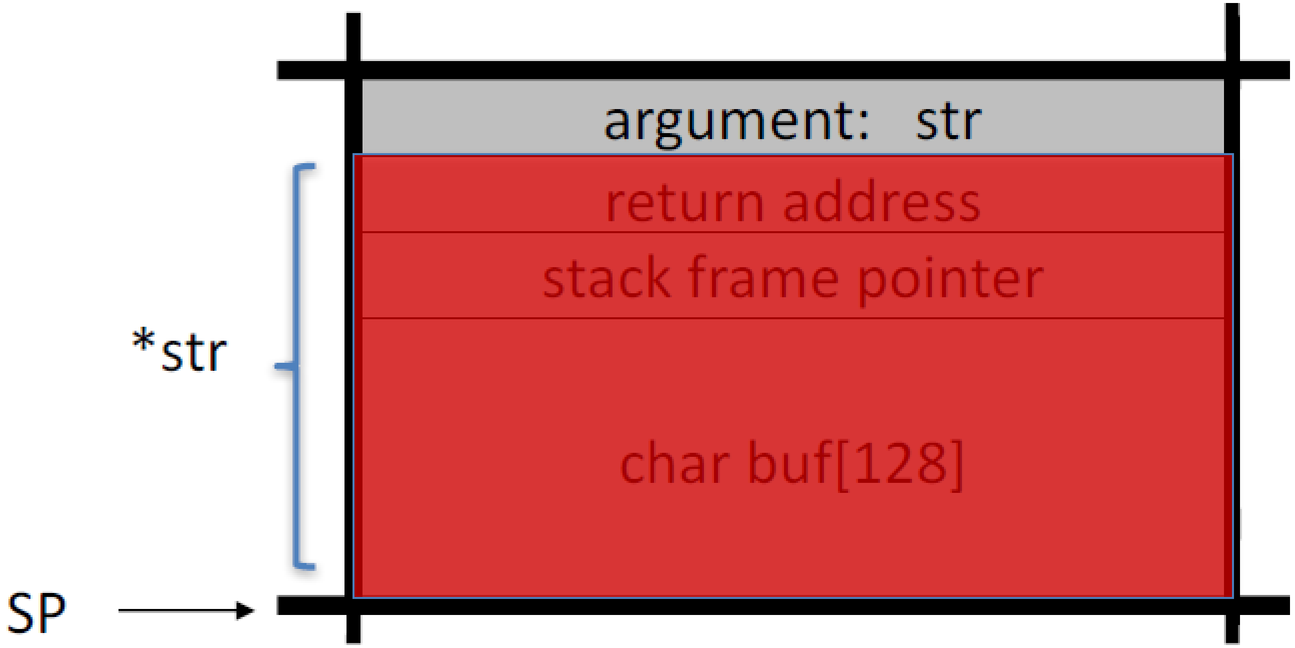
When `func()` is called stack looks like:



What are buffer overflows?

What if `*str` is 136 bytes long?

After `strcpy`:




```
void func(char *str)
{
   char buf[128];
   strcpy(buf, str);
   do-something(buf);
}
```

Problem:

no length checking in `strcpy()`

Basic stack exploit

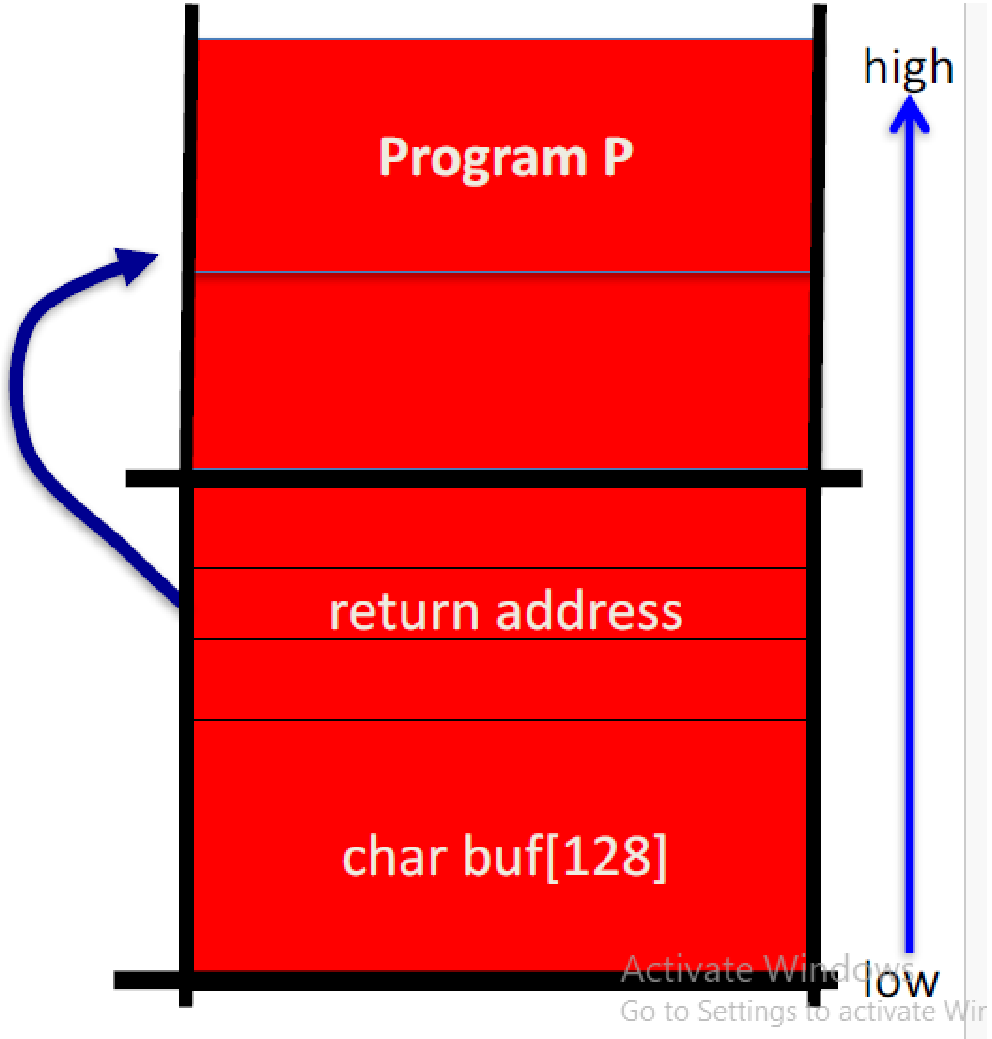
Suppose `*str` is such that after `strcpy` the stack looks like:

Program P: `exec("/bin/sh")`

(exact shell code by Aleph One)

When `func()` exits, the user gets shell!

Note: attack code P runs *in stack*.



Acknowledgments

- Dr Dan Boneh, Stanford University