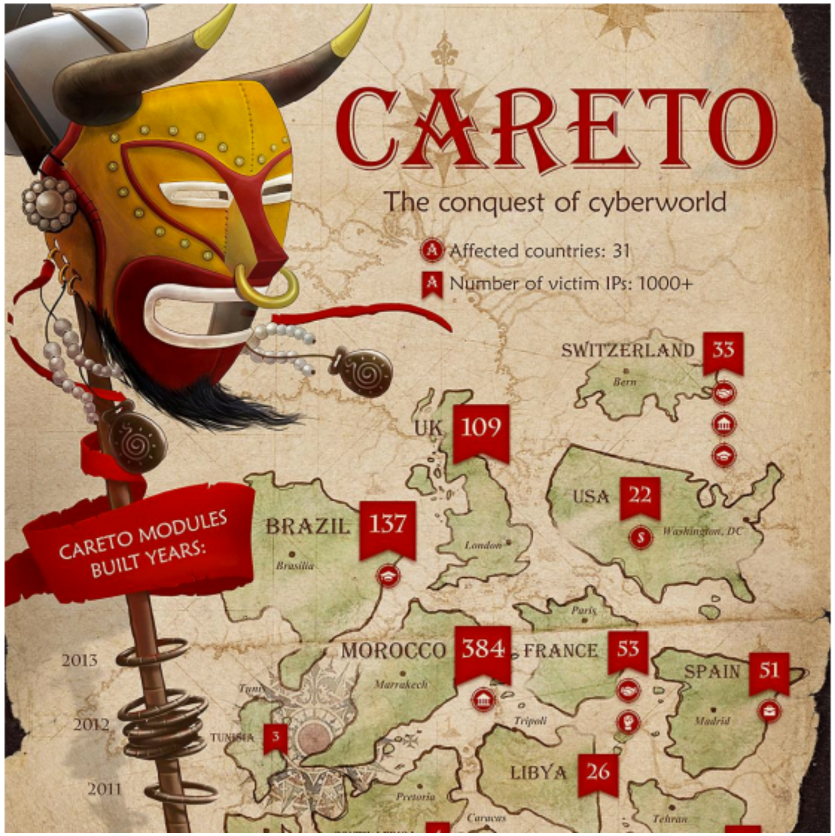
Memory corruption defenses

Lecture 10
Secure Programming

In the news



http://www.securelist.com/en/downloads/vlpdfs/unveilingthemask_v1.0.pdf

Where are we?

- We know how to exploit stack-based buffer overflow…
- …when all protection mechanisms are disabled (welcome back to early 2000s)
- Today, we start looking at protection mechanisms (“exploit mitigation”) and ways around them

Defense mechanisms

- Fix the human factor
  - Educate programmers on how to avoid writing insecure code
  - Test programs with a security emphasis
- Secure library functions
  - Standard lib: strncpy, strncat
  - Libsafe: replacement of dangerous functions with versions that perform run-time boundary checks
    http://www.cs.bham.ac.uk/~covam/teaching/2012/secprog/libsafe.pdf
- Secure languages
  - Cyclone, CCured, memory-safe languages

    http://www.cs.umd.edu/~mwh/papers/cyclone-cuj.pdf

    http://www.cs.sunysb.edu/~rob/teaching/cse608-fa05/ccured_popl02.pdf

Exploit mitigations

- Proper defensive mechanisms are the right approach to the problem
- There may be cases when they are impractical
  - Cannot convert entire code base into memory safe language
  - There is no way to force all developers to know about security
  - Legacy code
- Orthogonal approach: make it harder for attackers to successfully exploit a vulnerability

What exploit mitigations?

What do we require for a successful stack-based buffer overflow?

- *Overwrite* the value of the return address stored on the stack and return from the function
- Get the *address* of the shellcode and store it in place of the saved return address
- So we can jump there (on the stack) and *execute* our shellcode

What exploit mitigations?

What do we require for a successful stack-based buffer overflow?

- *Overwrite* the value of the return address stored on the stack and return from the function
  - Stack protection
- Get the *address* of the shellcode and store it in place of the saved return address
  - Address space randomization
- So we can jump there (on the stack) and *execute* our shellcode
  - Non executable stack

Remember attack trees?

And return-into-libc

NON EXECUTABLE STACK

Non executable stack

- Mark the pages where the stack is allocated as non executable
- Does not block the overflow, but prevents the shellcode from being executed
- Implemented in various forms by all major OSes
  - OpenBSD W^X
  - Data Execution Prevention (DEP) in Windows XP SP2 and Windows Server 2003
  - ExecShield and PAX patches for Linux
- (Used to) interfere with some programs that legitimately execute data on stack (e.g., JITs)

Non executable stack - mechanism

- Idea: load stack in portion of memory that is not executable
- Implementation is of course highly OS and CPU dependant
- Relies on protection mechanisms in MMU
  - NX (or XD) bit
  - Others have more coarse mechanisms (i386: code segment limit - “line in the sand”)
- Relies on support from compiler, linker, and loader to ensure that stack is allocated in protected memory
- Complexity: shared libraries, signals, constructors and destructors, etc.
- Example: how OpenBSD implemented W^X (also on i386)

  http://www.openbsd.org/papers/ven05-deraadt/mgp00009.html
  http://marc.info/?l=openbsd-misc&m=105056000801065

Non executable stack - Linux

```
$ gcc stack.c
      –o stack
$ execstack -q stack
- stack

bffdf000-c0000000 rw-
p 00000000 00:00 0
[stack]
```

```
$ gcc stack.c
      –z execstack
      –o stack
$ execstack -q stack
X stack

bffdf000-c0000000
rwxp 00000000 00:00 0
[stack]
```

Bypassing non executable stack

- Non executable stack prevents us to execute code that we injected
- We still have the capabilities of
  - Modifying the stack arbitrarily
  - Jumping and executing to existing program locations that are marked as executable
- Can we combine these capabilities to execute arbitrary (or at least useful) code?

Bypassing non executable stack

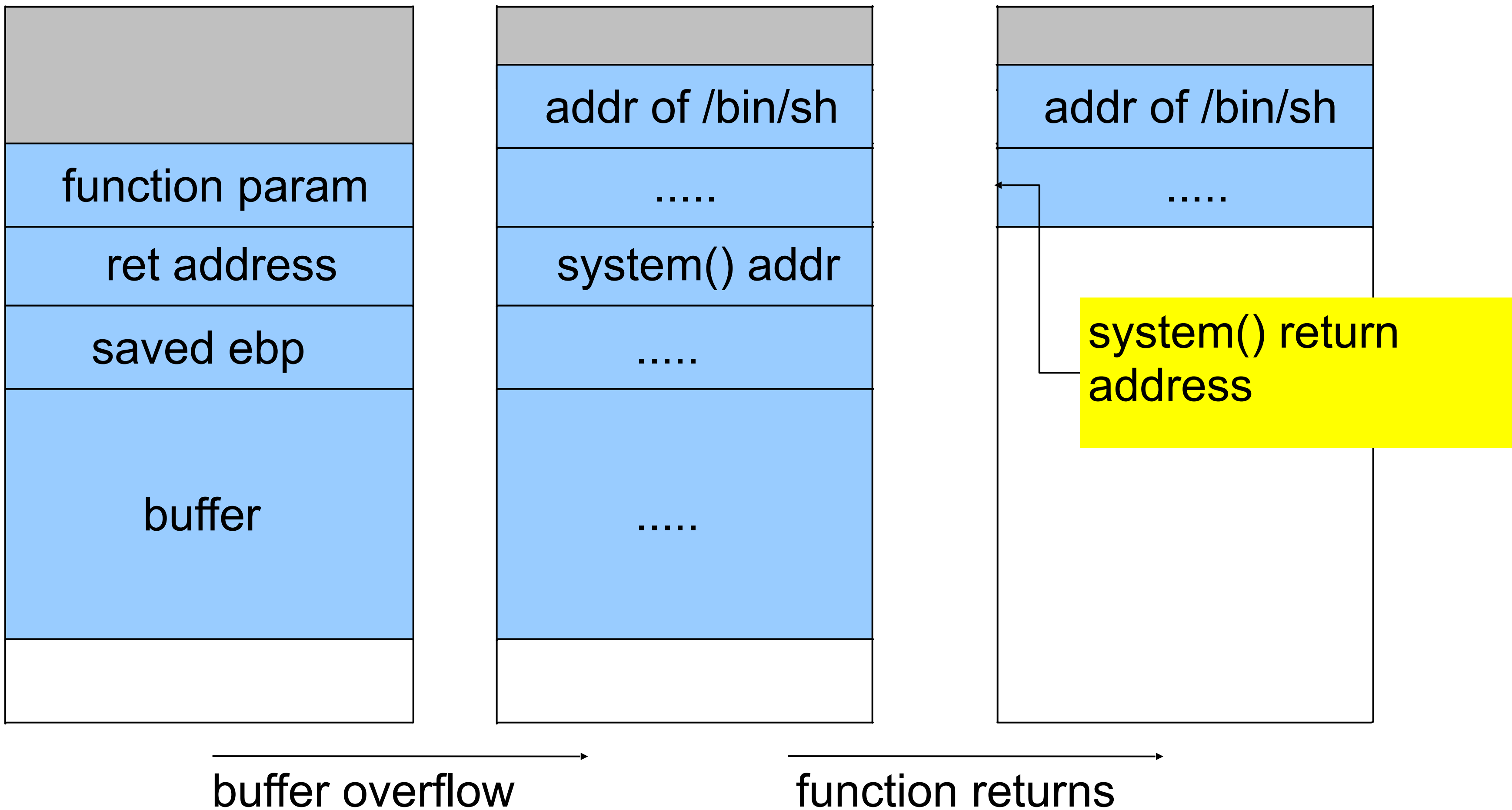
- Idea: call existing code, for example, library functions
- Attractive target is libc
  - Linked by practically all programs
  - Lots of interesting functions , e.g., `system()`, `exec*`
- Nergal, [The advanced return-into-lib(c) exploits: PaX case study](http://www.phrack.org/issues.html?issue=58&id=4), 2001
  http://www.phrack.org/issues.html?issue=58&id=4

Return-into-libc

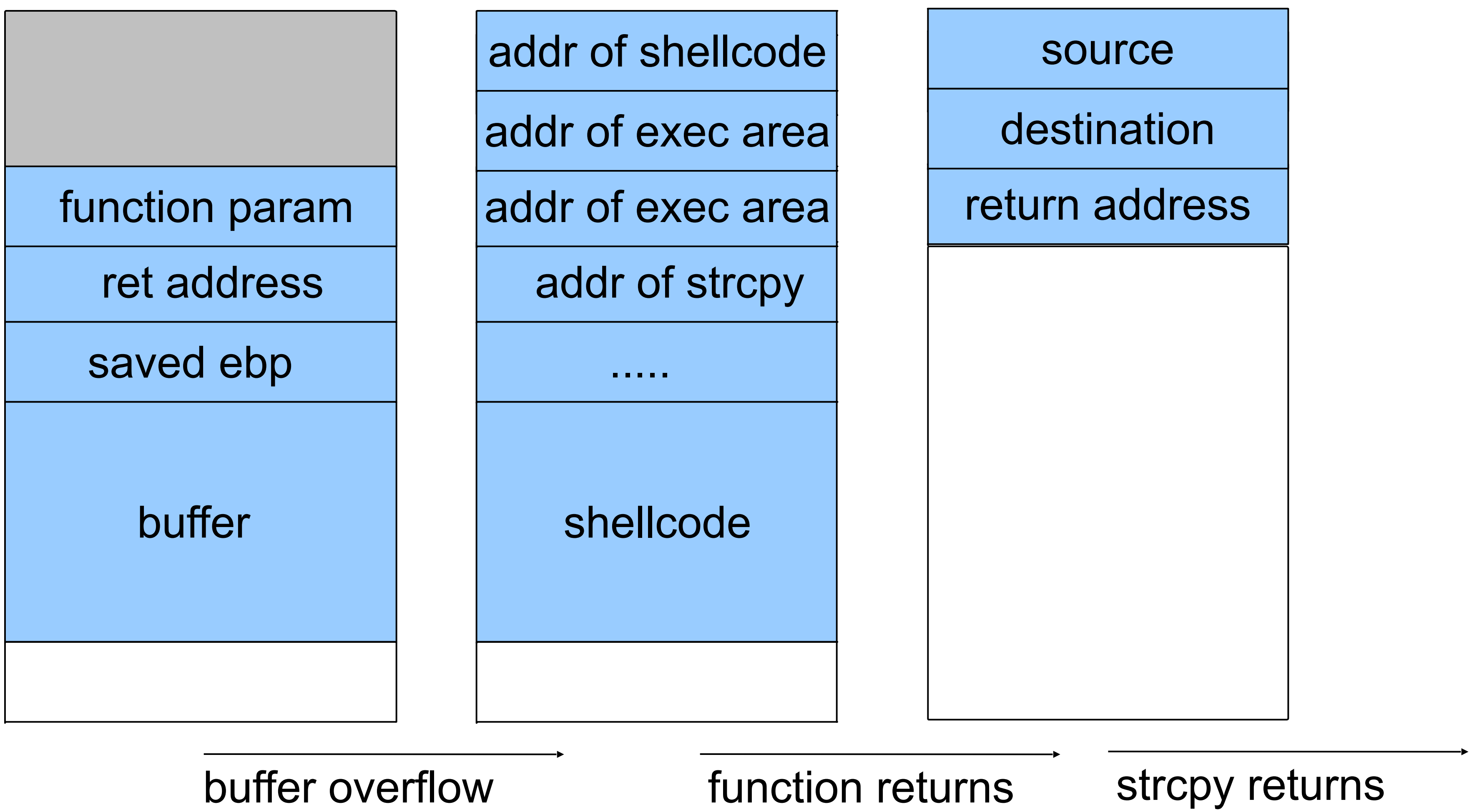
- Goal: we want to execute `system("/bin/sh")`
- Idea: overwrite the saved return address with the address of `system` function
- When the vulnerable function returns, `system` will start executing
- And it will look up in the stack its parameters

→ we need to set up the stack correctly

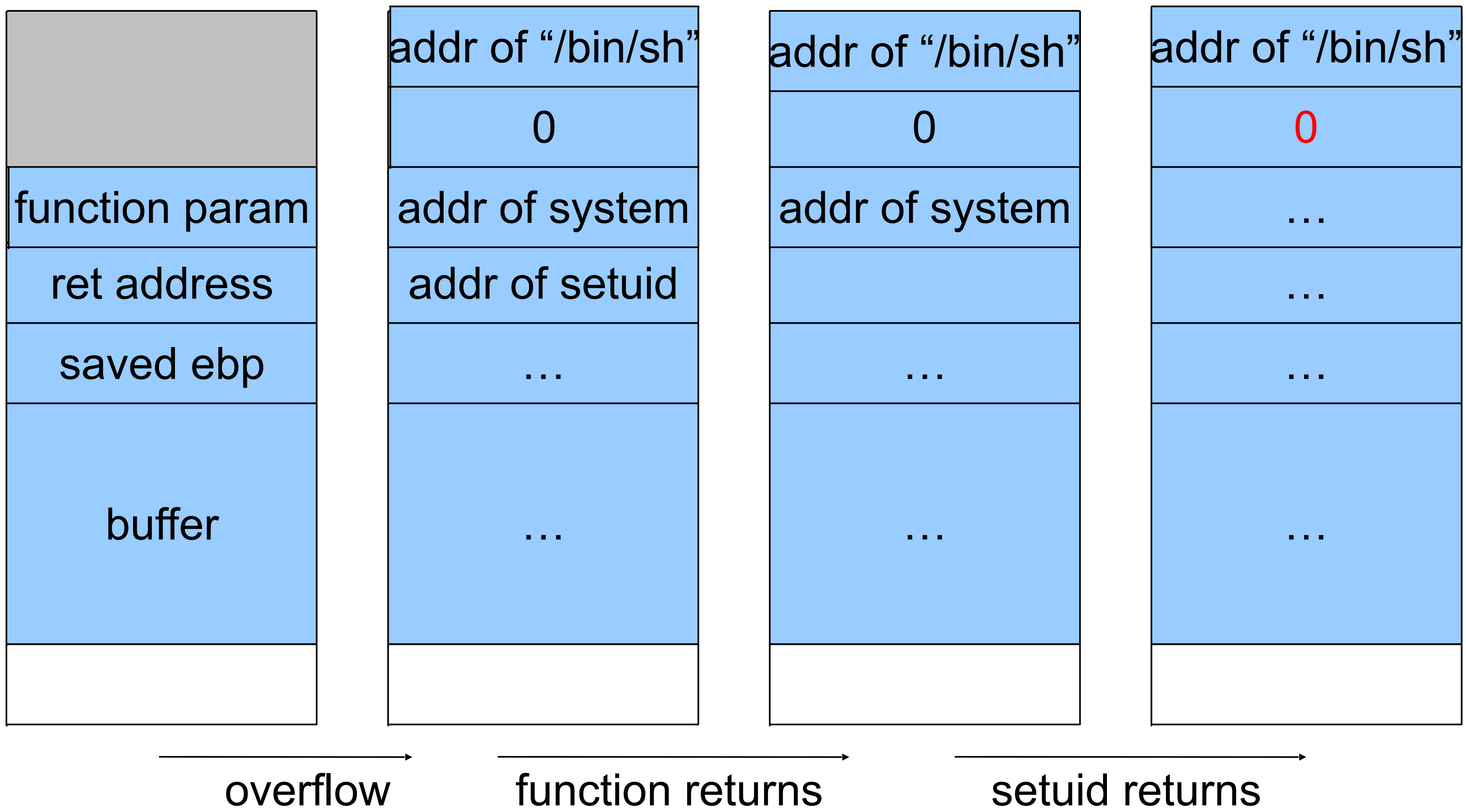
Return-into-libc: system



Return-into-libc: move shellcode



Chaining function calls



Chaining function calls

- This technique has limitations
  - Only works for calling 2 functions
  - The first function has exactly one parameter
- Can we do better?
  - We need to find ways to control the esp

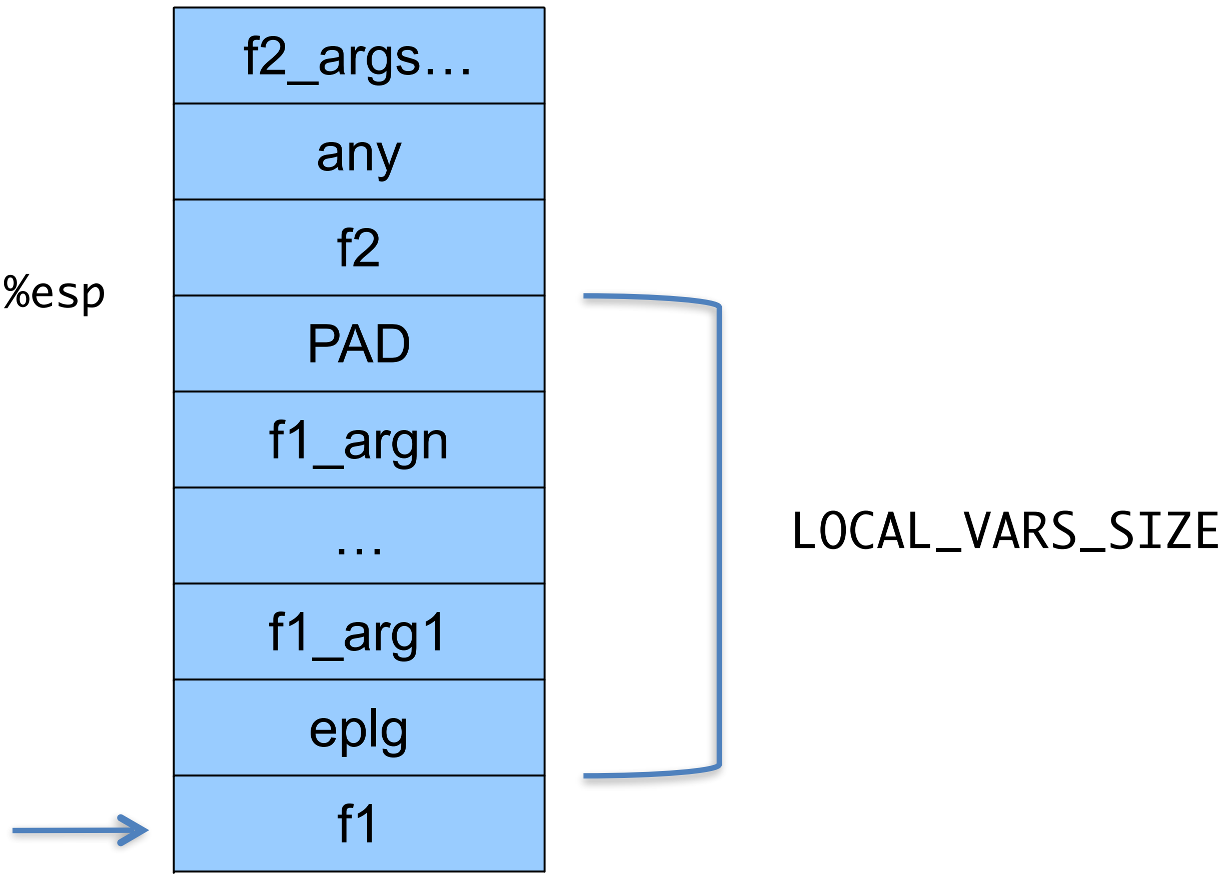
esp lifting

- Observation: possible to find sequences of instructions such as
  - eplg: `add $LOCAL_VARS_SIZE,%esp`
    `ret`
  - Typical function epilogue when code is compiled with -fomit-frame-pointer
- Enables us to mov $esp of fixed amount

esp lifting

- Goal: execute f1 and f2 in libc

```
eplg:
 add $LOCAL_VARS_SIZE,%esp
 ret
```



overwritten RET address

Next time

We continue looking at defenses:

- Stack protection
- Address space randomization