Technische Universität Braunschweig

IAS INSTITUTE FOR APPLICATION SECURITY

Vulnerabilities and Exploits

Vorlesung “Einführung in die IT-Sicherheit”

Prof. Dr. Martin Johns

Overview

- **Topic of the unit**
  - Vulnerabilities and Exploits
- **Parts of the unit**
  - Part #1: Basics of software security
  - Part #2: Stack and heap memory
  - Part #3: Stack and heap overflows
  - Part #4: Memory defenses

Vulnerabilities

- **Vulnerability** = exploitable flaw in a system or software
  - Basis for attacks on confidentiality, integrity or availability
  - Software bug not necessary a vulnerability
- **Where do vulnerabilities come from?**
  - Flaws in system design
  - Implementation errors
  - Misconfiguration
  - Inappropriate operation

Prevention of Vulnerabilities

- **Secure software development**
  - Security-aware design, implementation and operation
  - Requires time and expertise (↯cheap development)
  - Example: practice of secure programming
- **Common root-causes of vulnerabilities**
  - Inconsistent abstraction of semantics
  - Mixing of control structures and data

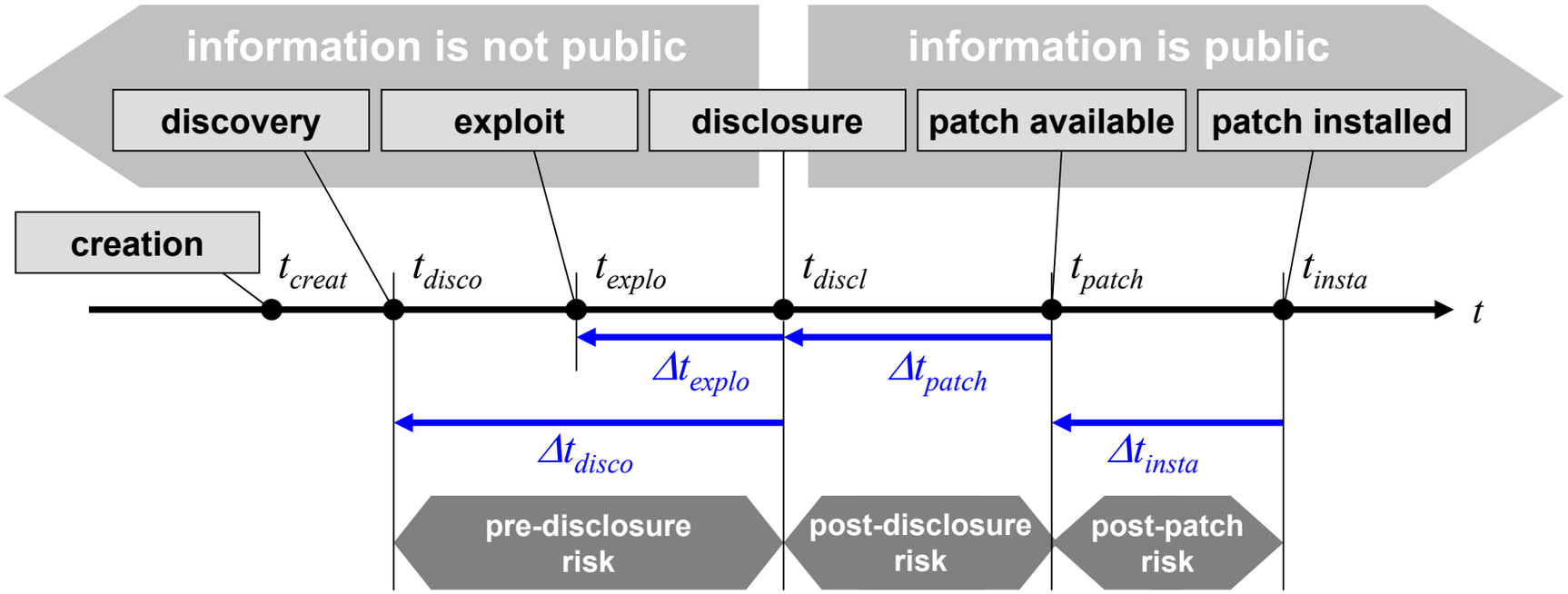
Exploits

- **Exploit** = program for exploiting a vulnerability
  - Usually manipulation of control flow
  - Bypass of authentication and access control
  - Proof of concept vs. malicious code
- **Common exploit phases**
  - Injection of code or data
  - Manipulation of control flow
  - Elevation of privileges
- No restrictions on exploit design

Exploit toolkit: Metasploit

Vulnerability Lifecycle

- **Different points in the "life" of a vulnerability**
  - Pivotal point: public disclosure of vulnerability
  - Vulnerability prior to its disclosure denoted as "zero-day"



(Frei et al., WEIS 2009)

Some Vulnerability Types

Fun with Memory

Stack overflows
Heap overflows
Integer overflows

→ Today's security lecture

Fun with Concurrency

Race conditions
Deadlocks
Livelocks

→ Operating system course

Fun with Validation

Cross-site Scripting
SQL injection
Remote file inclusion

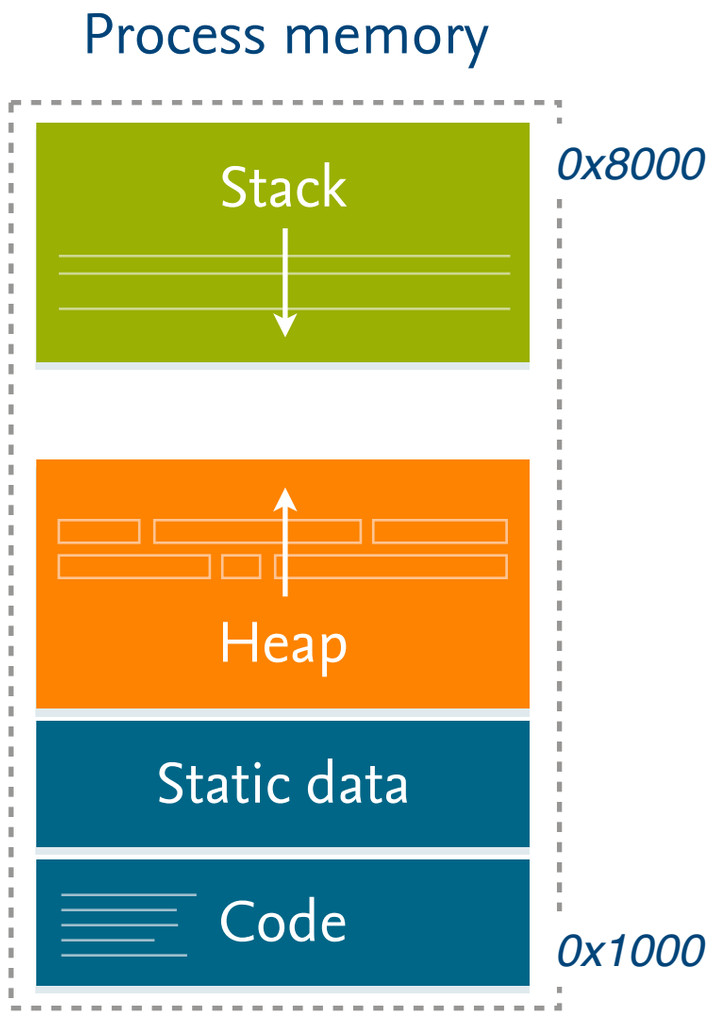
→ Previous lecture on web security

Overview

- **Topic of the unit**
  - Vulnerabilities and Exploits
- **Parts of the unit**
  - Part #1: Basics of software security
  - Part #2: Stack and heap memory
  - Part #3: Stack and heap overflows
  - Part #4: Memory defenses

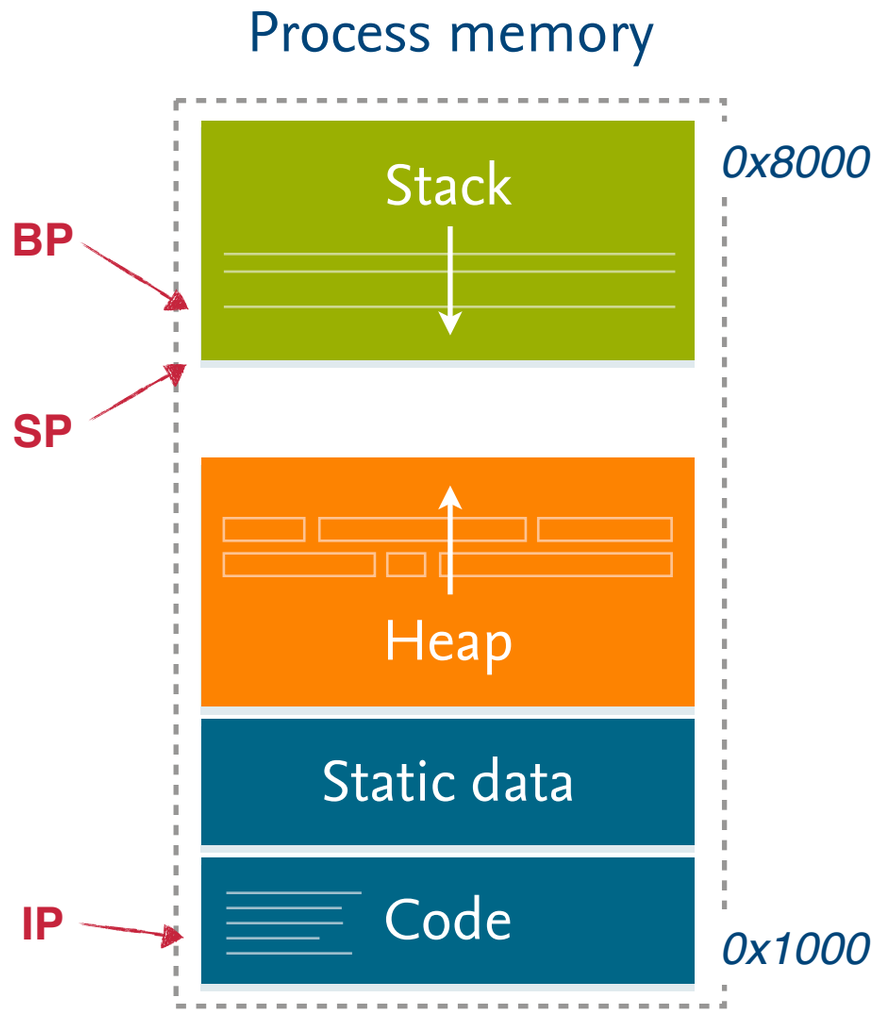
Program Execution

- **Execution of a program**
  - Loading of program into memory
  - Different memory segments for code, static data and dynamic data
  - Execution of machine code instructions in code segment
- **Special regions for dynamic data**
  - Stack segment (LIFO access)
  - Heap segment (arbitrary access)



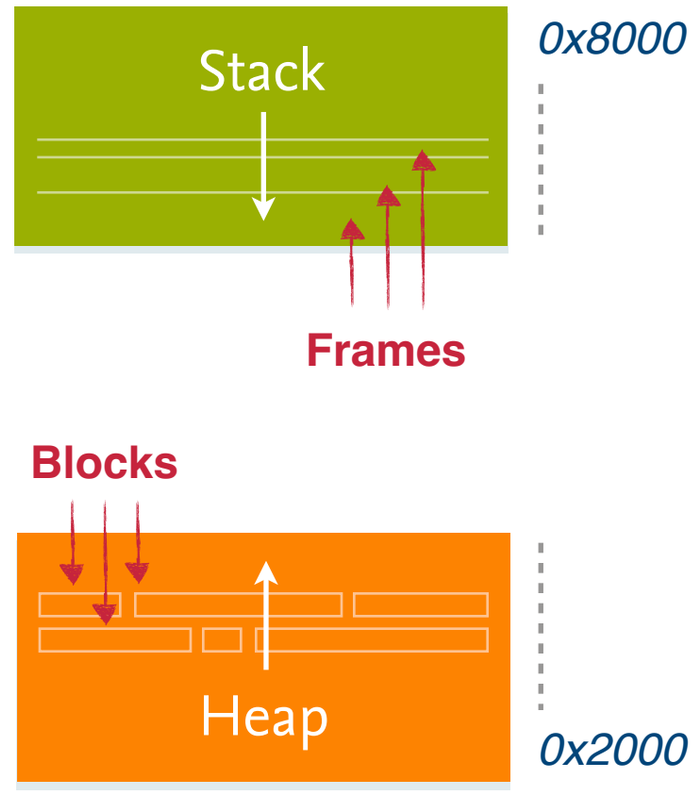
CPU Registers

- **Relevant CPU registers**
  - PC / IP = Instruction pointer
    points to current instruction
  - SP = Stack pointer
    points to end of stack
  - BP = Base pointer
    points to current stack frame
- **Other registers depending on CPU architecture**



Heap and Stack

- **Stack segment**
  - Temporary storage for functions
  - LIFO access to so-called frames
  - Growth from high to low addresses



- **Heap segment**
  - Allocation of arbitrary blocks
  - General-purpose memory
  - Growth from low to high addresses

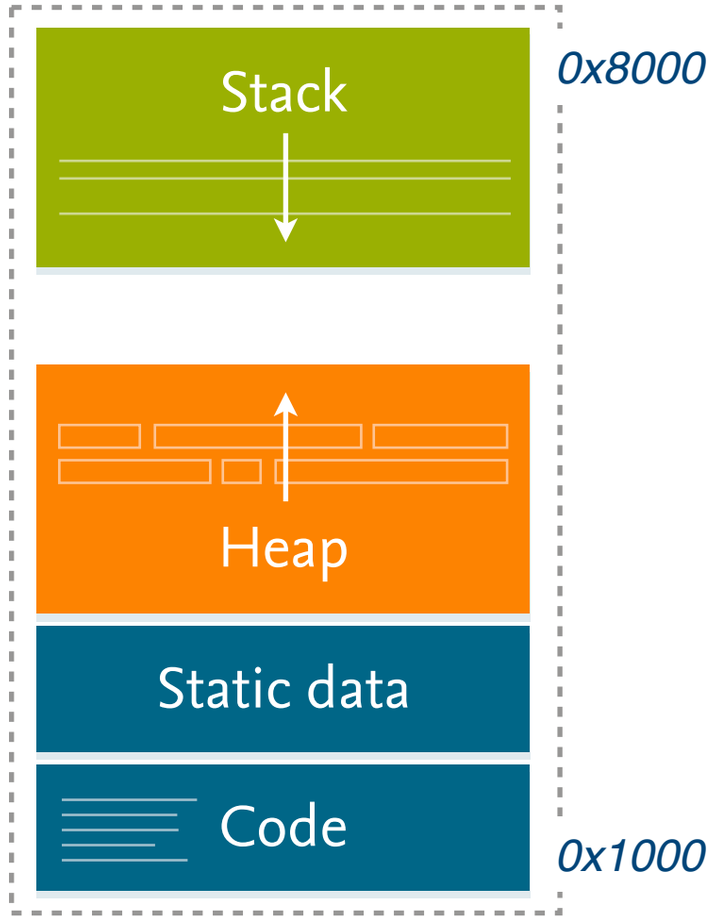
Stack Usage

A Program Snippet

C code snippet

```
int main(int argc, char **argv) {
    int a;
    char *c;
    c = malloc(10);
    foobar(a, c);
    free(c);
    return a;
}

void foobar(int a, char *c) {
    int x, y;
    c++;
    return;
}
```

Process memory

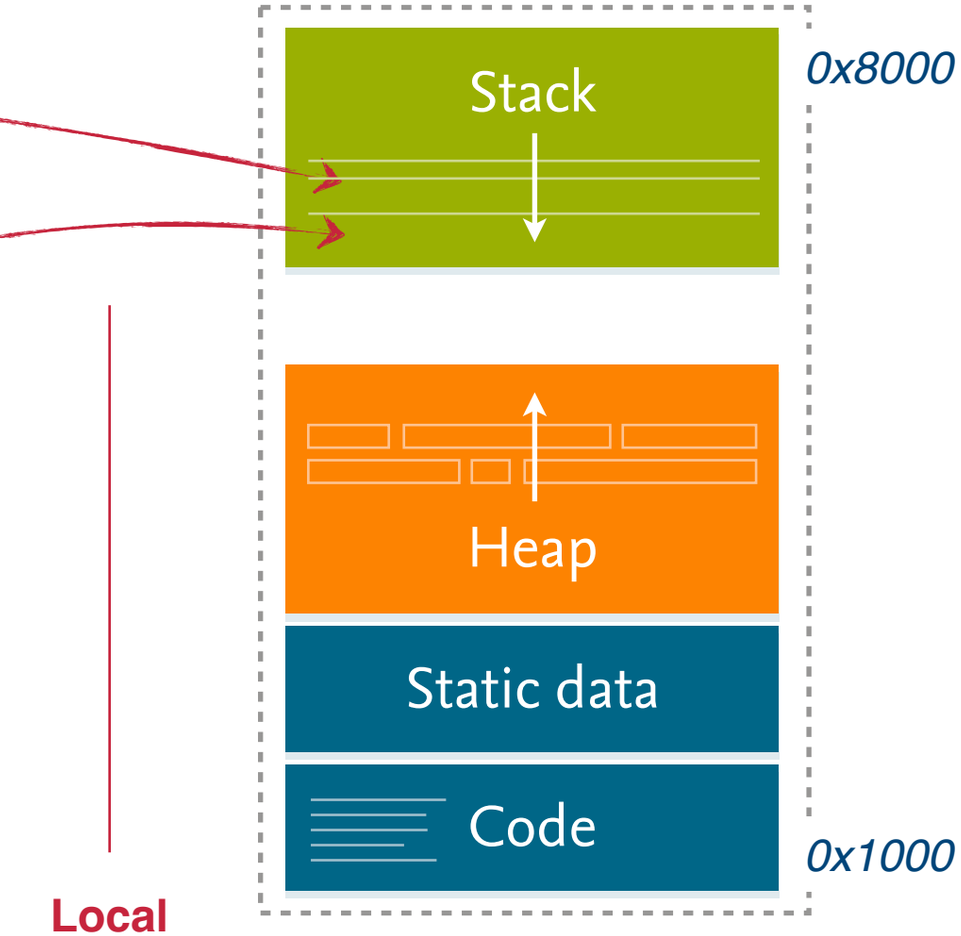


A Program Snippet

C code snippet

```
int main(int argc, char **argv) {
    int a;
    char *c;
    c = malloc(10);
    foobar(a, c);
    free(c);
    return a;
}

void foobar(int a, char *c) {
    int x, y;
    c++;
    return;
}
```

Process memory



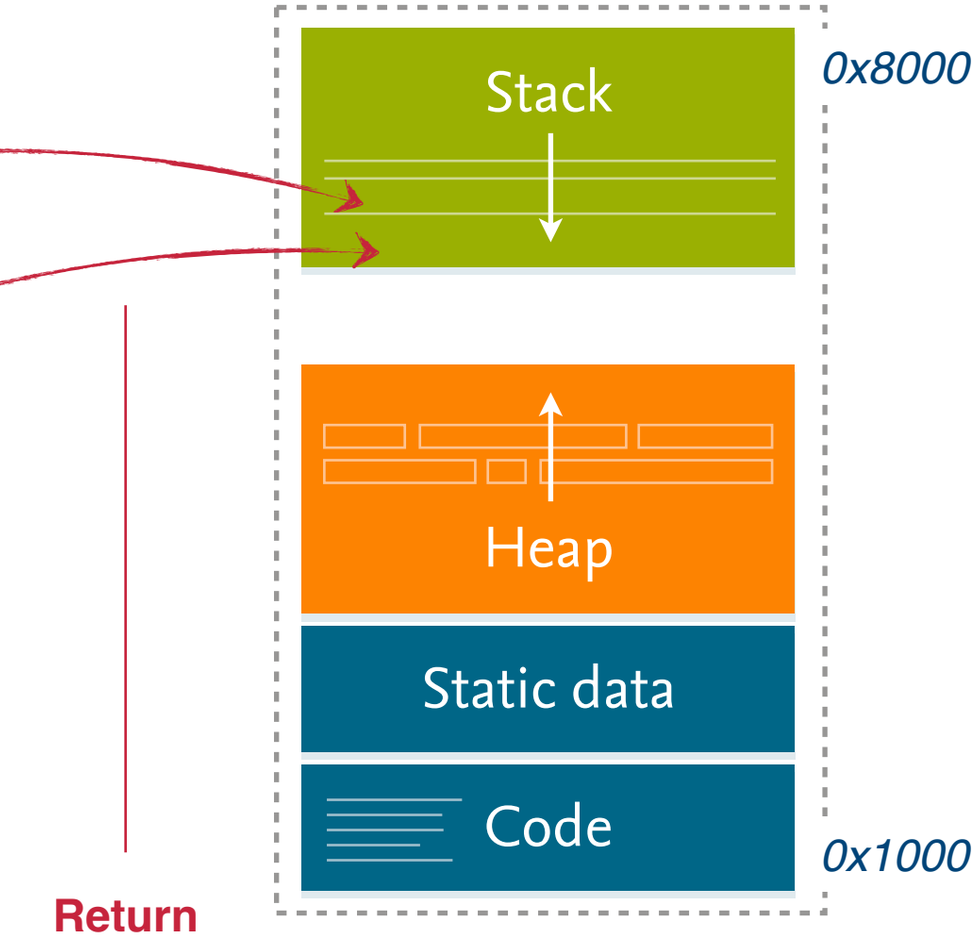
Local variables

A Program Snippet

C code snippet

```
int main(int argc, char **argv) {
    int a;
    char *c;
    c = malloc(10);
    foobar(a, c);
    free(c);
    return a;
}

void foobar(int a, char *c) {
    int x, y;
    c++;
    return;
}
```

Process memory



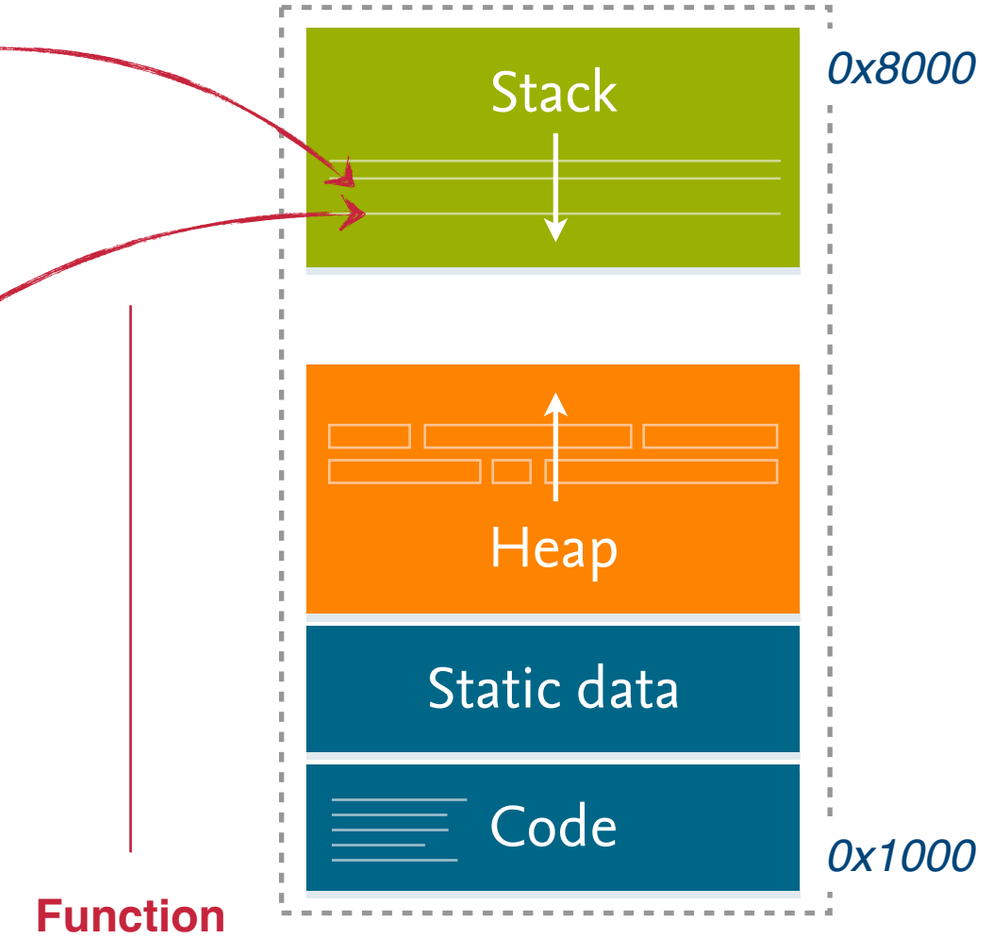
Return addresses

A Program Snippet

C code snippet

```
int main(int argc, char **argv) {
    int a;
    char *c;
    c = malloc(10);
    foobar(a, c);
    free(c);
    return a;
}

void foobar(int a, char *c) {
    int x, y;
    c++;
    return;
}
```

Process memory



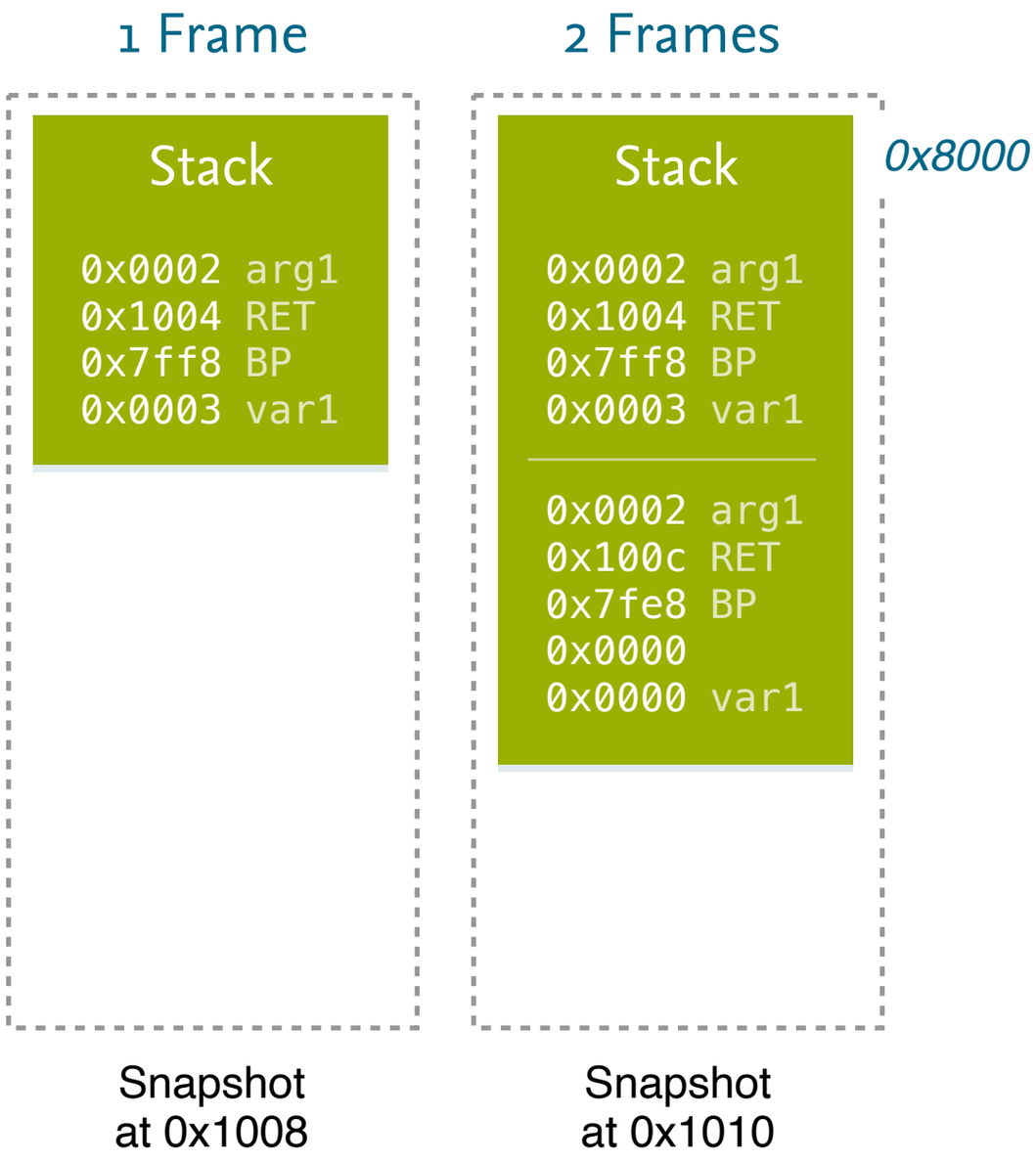
Function arguments

Stack Frames

C code snippet

```
0x1000:  int main() {
0x1002:    foo(2);
0x1004:  }

0x1006:  void foo(int arg1) {
0x1008:    int var1 = arg1 + 1;
0x100a:    bar(arg1);
0x100c:  }

0x100e:  void bar(int arg1) {
0x1010:    char var1[4];
0x1012:  }
```

Example of stack frames on artificial 16 bit architecture



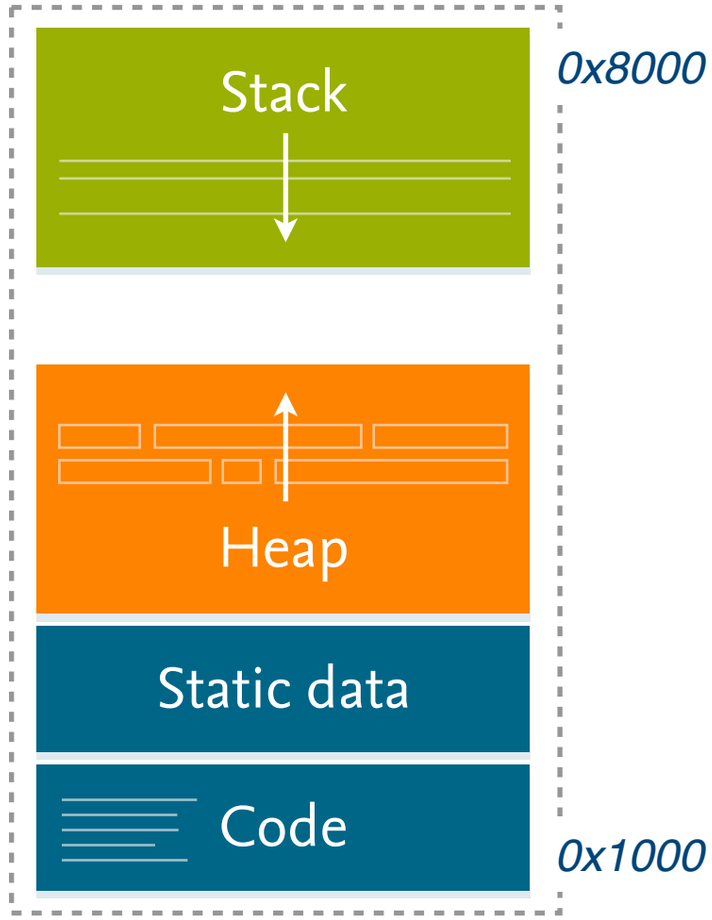
Heap Usage

A Program Snippet

C code snippet

```
int main(int argc, char **argv) {
    int a;
    char *c;
    c = malloc(10);
    foobar(a, c);
    free(c);
    return a;
}

void foobar(int a, char *c) {
    int x, y;
    c++;
    return;
}
```

Process memory

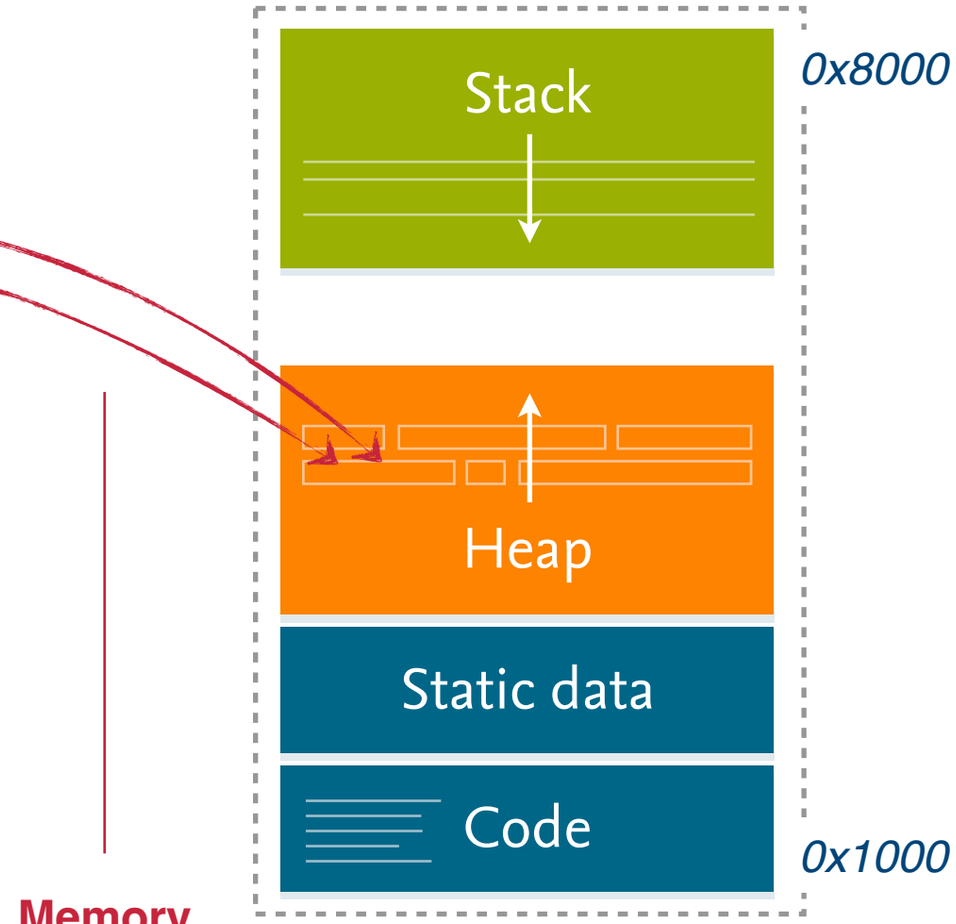


A Program Snippet

C code snippet

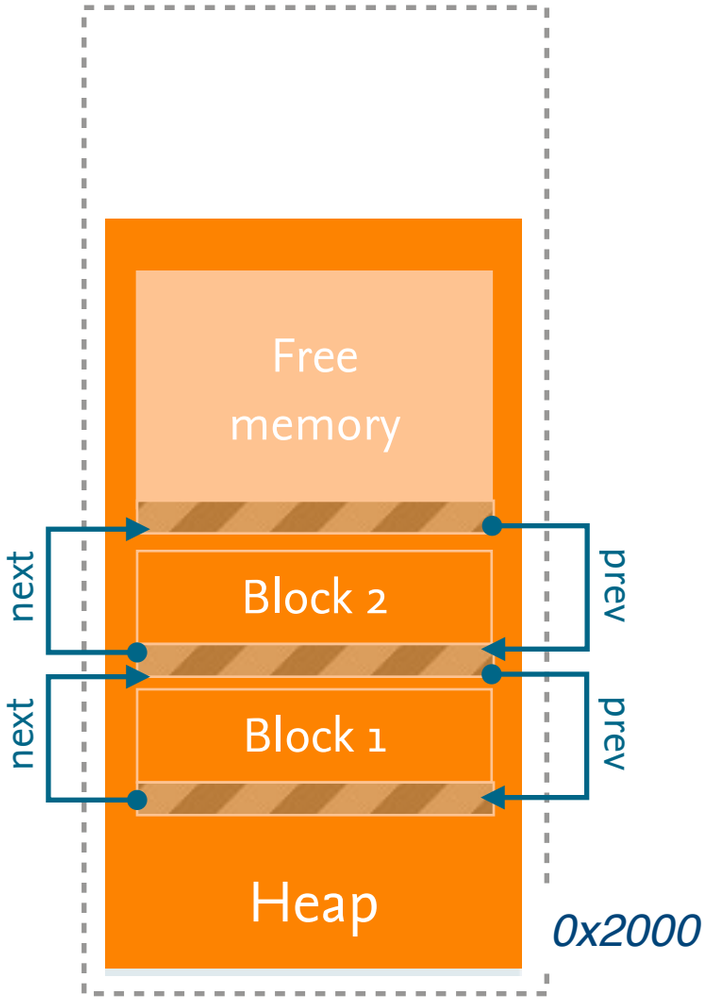
```
int main(int argc, char **argv) {
    int a;
    char *c;
    c = malloc(10);
    foobar(a, c);
    free(c);
    return a;
}

void foobar(int a, char *c) {
    int x, y;
    c++;
    return;
}
```

Process memory

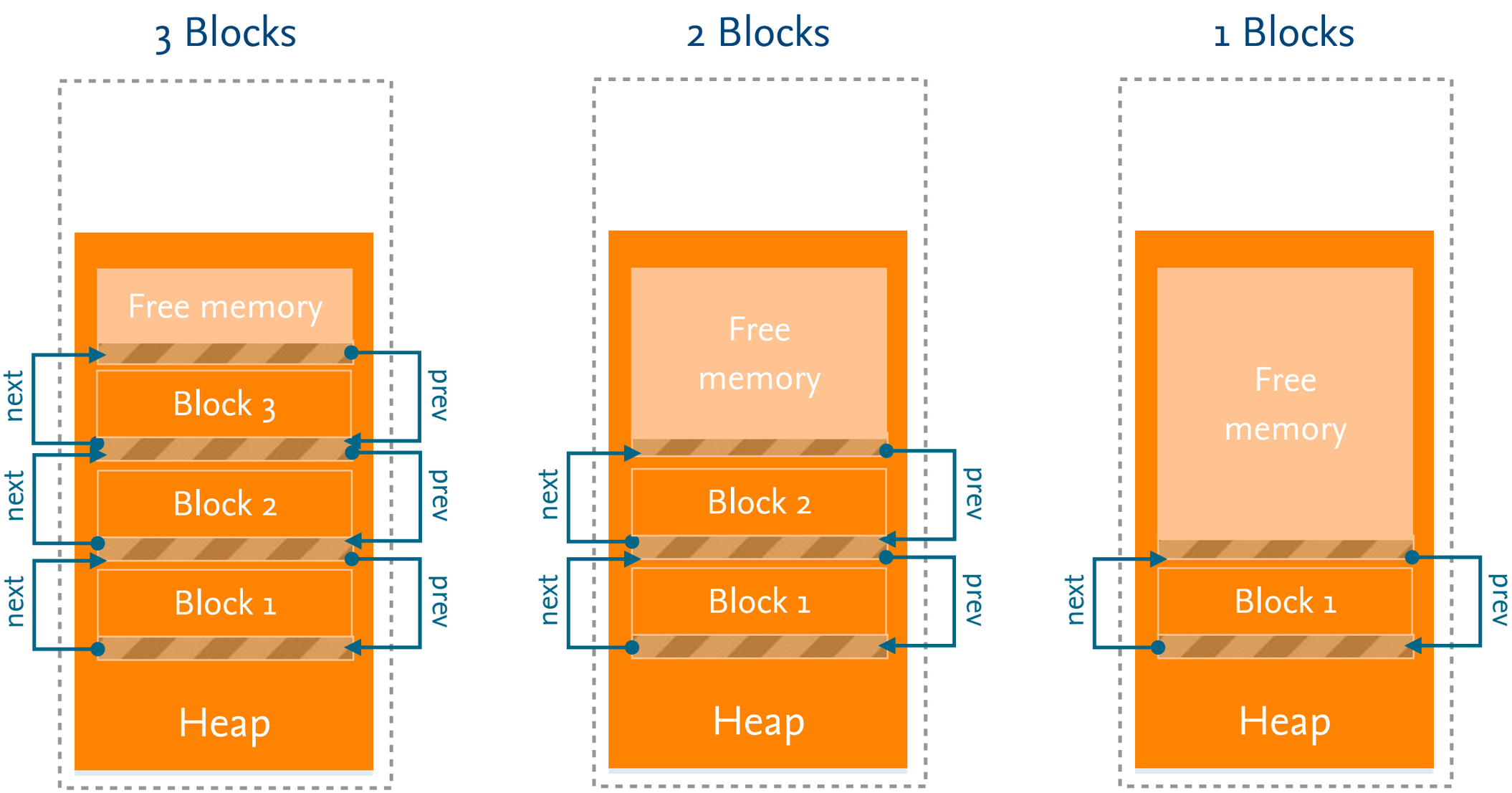


Heap Memory

- **Various heap implementations**
  - Often blocks maintained in linked list
  - Metadata at start of each block
- **Allocation of a heap block**
  - Split block from free memory block
- **Deallocation of a heap block**
  - Merge block with free memory block



Allocation and Deallocation



Some Pseudo-Code

- **Allocation code snippet**
  - New block is allocated
  - Successor is updated
  - Predecessor is updated

Snippet from alloc()

```
new_b = alloc_block();

old_b->next->prev = new_b;
new_b->next = old_b->next;

old_b->next = new_b;
new_b->prev = old_b;
```

- **Deallocation code snippet**
  - Block is removed from list
  - Neighbors are updated

Snippet from free()

```
b->next->next->prev = b->next->prev;
b->next = b->next->next;
```

Simplified code. See article by Felix ‘FX’ Lindner “A Heap of Risk” for a better example.

Overview

- **Topic of the unit**
  - Vulnerabilities and Exploits
- **Parts of the unit**
  - Part #1: Basics of software security
  - Part #2: Stack and heap memory
  - Part #3: Stack and heap overflows
  - Part #4: Memory defenses

Buffer Overflows

- **Buffer** = fixed-length array (of one data type)
  - … on the stack: local variable

```
char buffer[4];
```

  - … on the heap: pointer to block

```
char *ptr = malloc(4);
```

- **Buffer overflow** = writing past the end of a buffer
  - Following data and metadata overwritten
  - Written data controlled by attacker
- **Exploitability dependent on different factors**
  - Location of memory (heap or stack)
  - CPU architecture and memory layout

Why do overflows happen?

- **Insufficient allocation of memory**
  - Missing awareness of overflow problem
  - Unbounded string functions: gets, strcpy, sprintf, ...
- **Incorrect calculation of size or location**
  - Integer and arithmetic overflows
  - Type conversions (casting)
  - Dangerous string functions: strncpy, strncat, ...

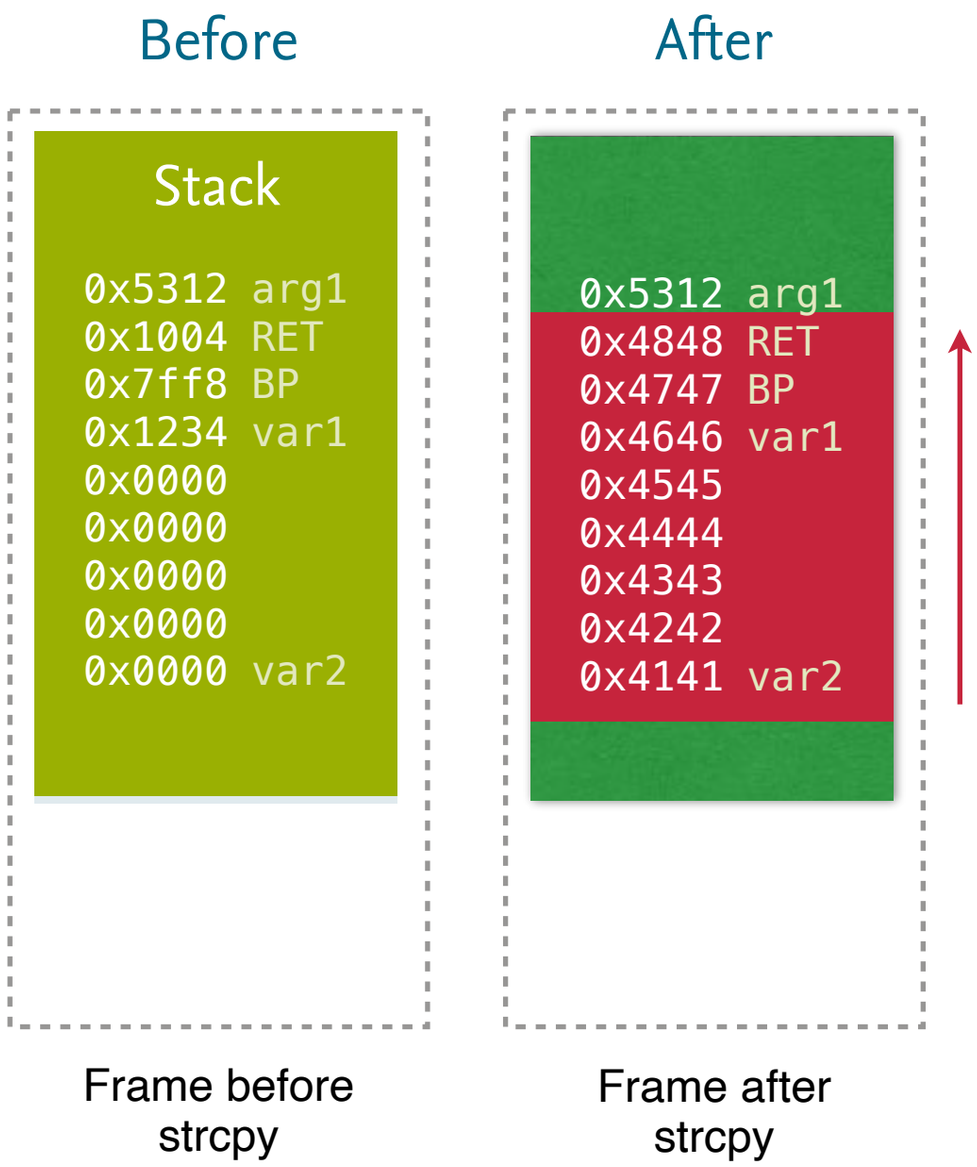
Stack Overflows

Vulnerable C code

```
void func(char *arg1) {
   int var1 = 0x1234;
   char var2[10];
   strcpy(var2, arg1);
}
```

Let's overflow it!

```
func("AABBCCDDEEFFGGHH");
```



Stack Overflows

- **Exploitation of a stack overflow**
  - Overwriting of current and previous stack frames
  - Manipulation of local variables and return addresses
- **Manipulation of local variables**
  - Redirection of control flow by variable assignments
- **Manipulation of return address**
  - Injection of executable code (so-called shellcode)
  - Redirection of return address to injected code

Overflow of Local Variable

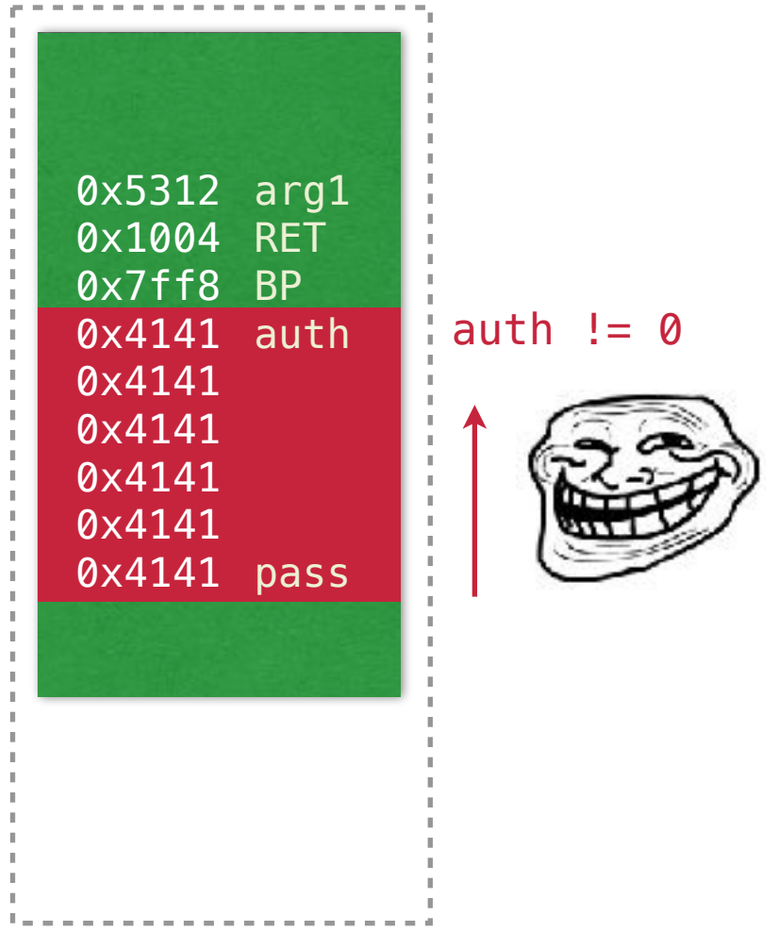
Vulnerable C code

```
void func(char *str) {
   int auth = 0;
   char pass[10];
   strcpy(pass, str);

   if (!strcmp(pass, “s3cr3t”))
       auth = 1

   if (auth != 0)
       printf(“success”);
   else
       printf(“failed”);
}
```

Exploit

```
func(“AAAAAAAAAAAA”);
```

Frame



Frame after strcpy

Overflow of Return Address

Vulnerable C code

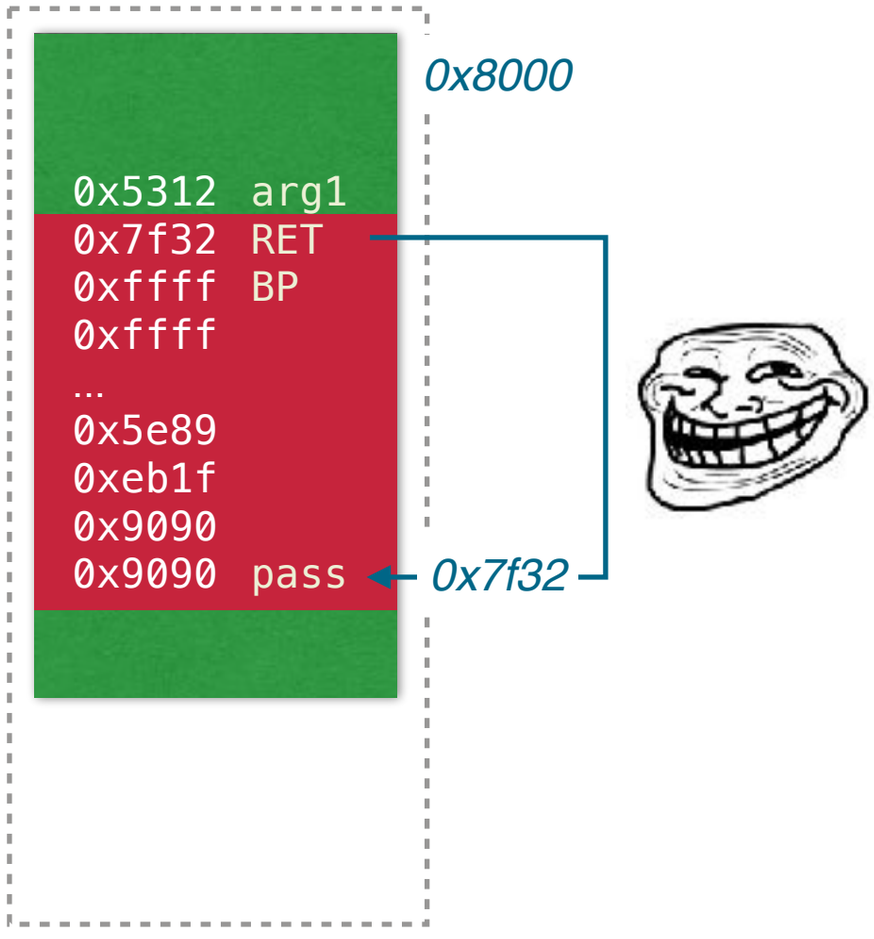
```
void func(char *str) {
   char pass[128];
   strcpy(pass, str);

   if (strcmp(pass, “s3cr3t”))
       return;
…
```

Exploit

```
/* NOPs + Shellcode + new RET */
func(“\x90\x90 … \x7f\x32”);
```

Control flow redirected to attacker’s code

Frame



Frame after strcpy

Shellcode

- **Shellcode** = code used after exploitation of a vulnerability
  - Originally used for launching a command shell
  - Today various types: bindshell, staged shellcode, ...
  - Often constraints on content of shellcode, e.g. null-free

Simple Linux shellcode by Steve Hanna

```
        jmp short end
start:  pop ebx              ; get address of string
        xor eax, eax
        mov [ebx+7 ], al     ; put a NULL where N is in the string
        mov [ebx+8 ], ebx    ; put address of string to where the AAAA is
        mov [ebx+12], eax    ; put 4 null bytes to where the BBBB is
        mov al, 0x0b         ; execve is syscall 0x0b
        lea ecx, [ebx+8]     ; load the address of where the AAAA was
        lea edx, [ebx+12]    ; load the address of where the BBBB was
        int 0x80             ; call the kernel: execve("/bin/sh", AAAA, BBBB);
end:    call start
        db '/bin/shNAAAABBBB'
```

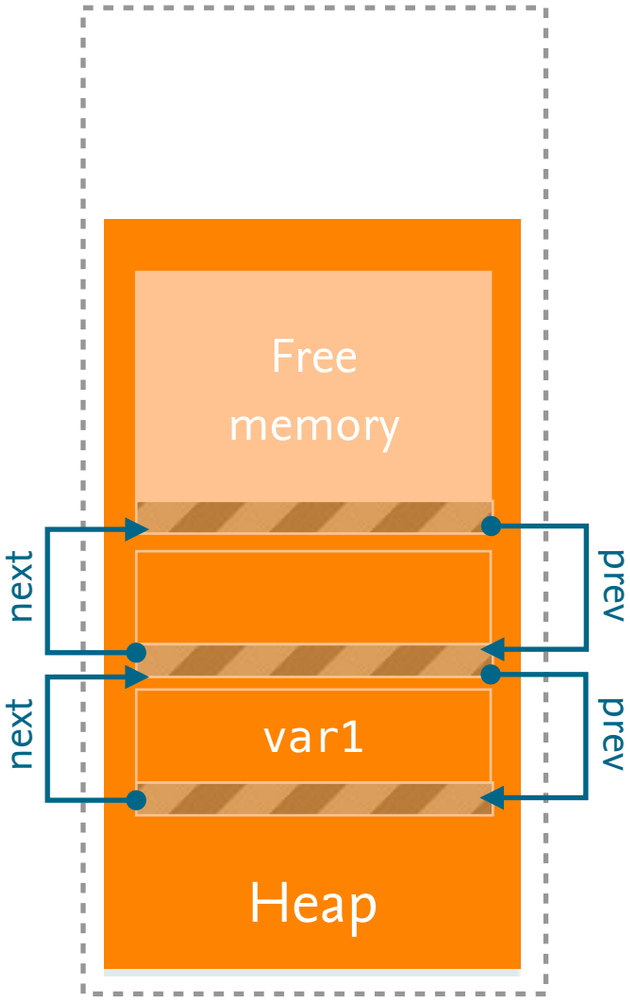
Heap Overflow

Vulnerable C code

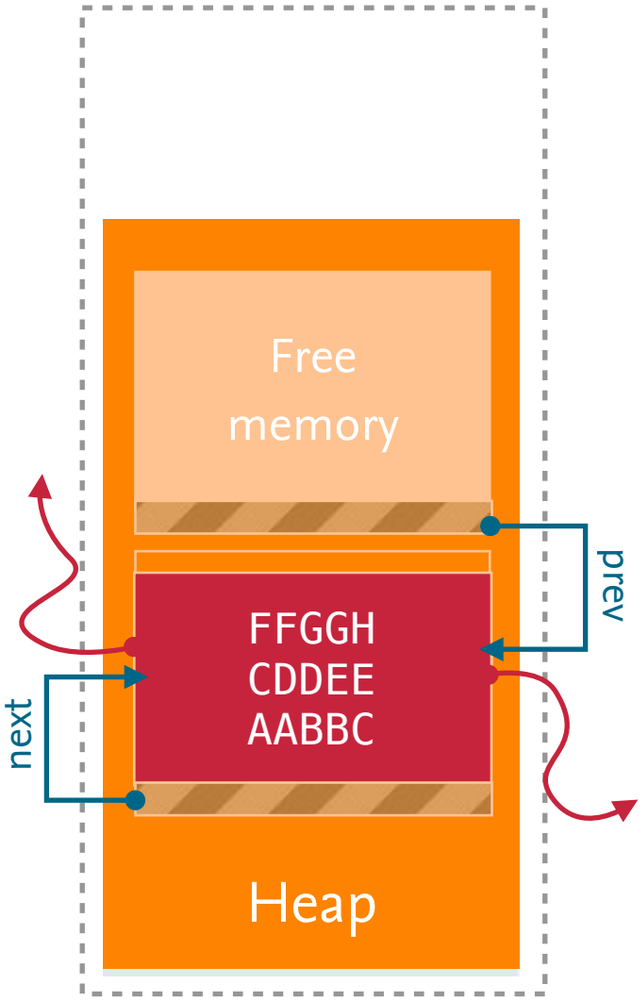
```
void func(char *arg1) {
   char *var1;
   var1 = malloc(4);
   strcpy(var2, arg1);
}
```

Let's overflow it!

```
func("AABBCCDDEEFFGGH");
```



Heap before strcpy



Heap after strcpy

Heap Overflow

- **Exploitation of a heap overflow**
  - Overwriting of next heap block and its metadata
  - Manipulation of list pointers only (no jump address)
  - Exploitation highly implementation-dependent
- **Classic unlink attack**
  - Manipulation of list pointers to write to arbitrary address
  - Triggered on merging (unlink function)

Unlink Attack

Snippet from free()

```
b->next->next->prev = b->next->prev;
```

Target address ← Value

Modification of data at arbitrary memory location

- **Redirection of control flow by …**
  - Manipulation of exception handlers
  - Manipulation of maintenance functions
  - Manipulation of data on the heap
  - Difficulty: Exact memory addresses need to be known

Overview

- **Topic of the unit**
  - Vulnerabilities and Exploits
- **Parts of the unit**
  - Part #1: Basics of software security
  - Part #2: Stack and heap memory
  - Part #3: Stack and heap overflows
  - Part #4: Memory defenses

Defenses

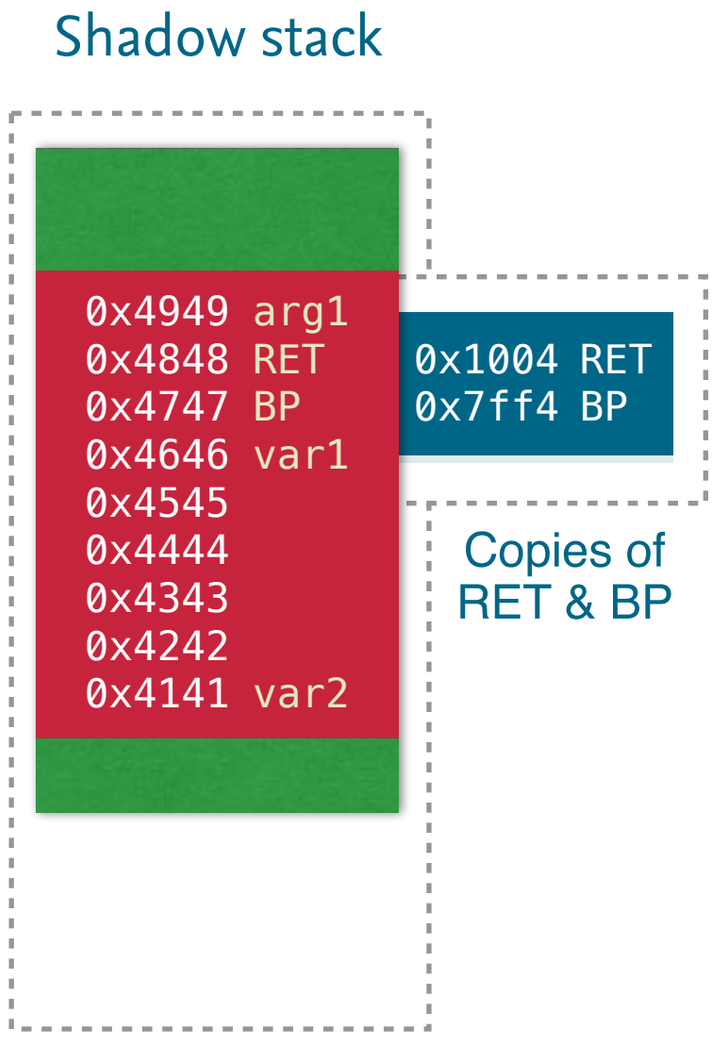
- **Static measures against overflow attacks**
  - Replacement of insecure functions
  - Boundary checks at compile-time
- **Dynamic measures against overflow attacks**
  - Boundary checks at run-time
  - Shadowing of metadata
  - Access control of memory segments
  - Randomization of memory layout

Secure Functions

- **Replacement of insecure functions**
  - Explicit and implicit boundary checks
  - Consistent interfaces to memory access
  - Removal of inherently insecure functions
- **Some implementations**
  - libsafe: Replacement of string and memory functions
  - strlcat/strlcpy: Consistent string operations from OpenBSD
  - Recent patches and extensions to GCC and LibC

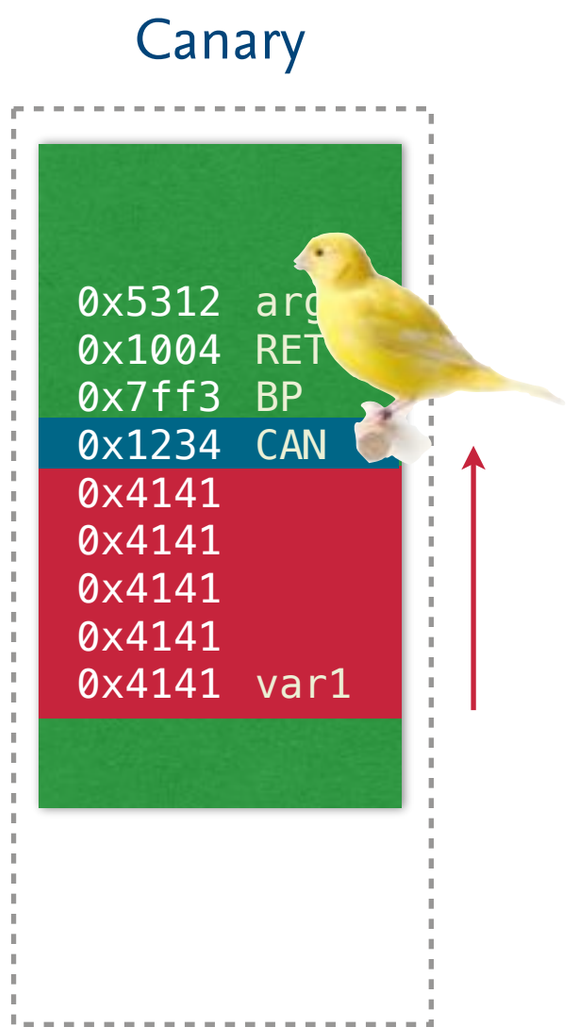
Shadow Stack

- **Shadowing of metadata**
  - Maintenance of additional stack
  - Duplication of RET and BP values
  - Comparison at function epilog
  - Inconsistence aborts program
- **Run-time overhead**
  - Interception of stack usage
- **Hardens exploitation of stack overflows**



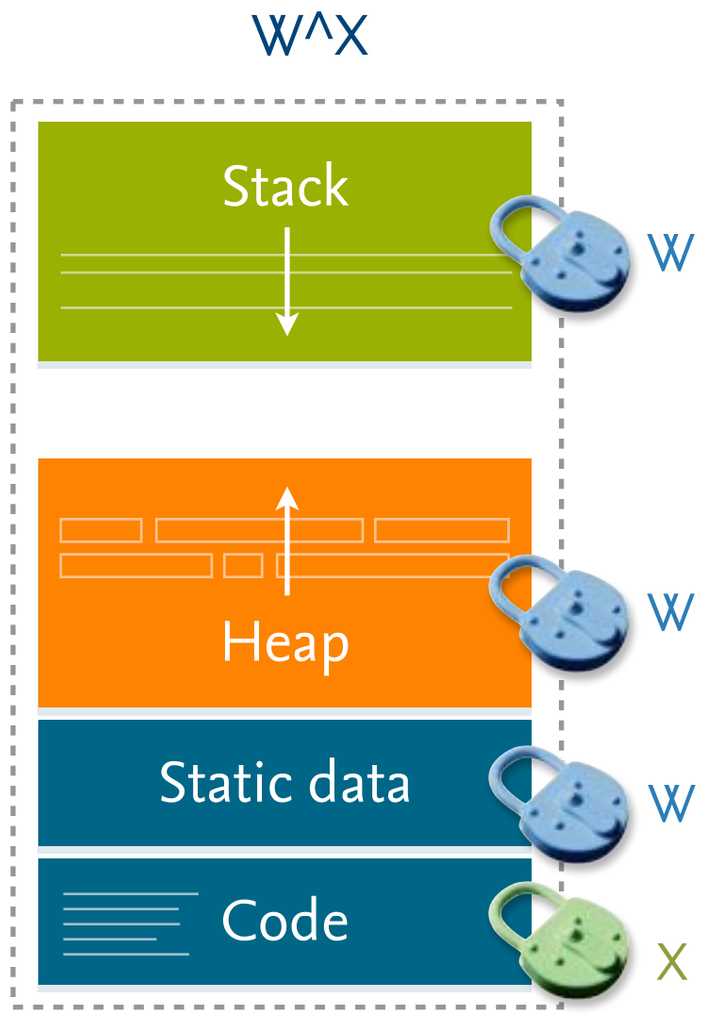
Canaries

- **Boundary check added at compile-time**
  - Re-compilation of software necessary
- **Canary safeguard**
  - Guard value stored in each stack frame
  - Value positioned before RET and BP
  - Comparison at function epilog
  - Manipulation of value aborts program
- **Hardens exploitation of stack overflows**



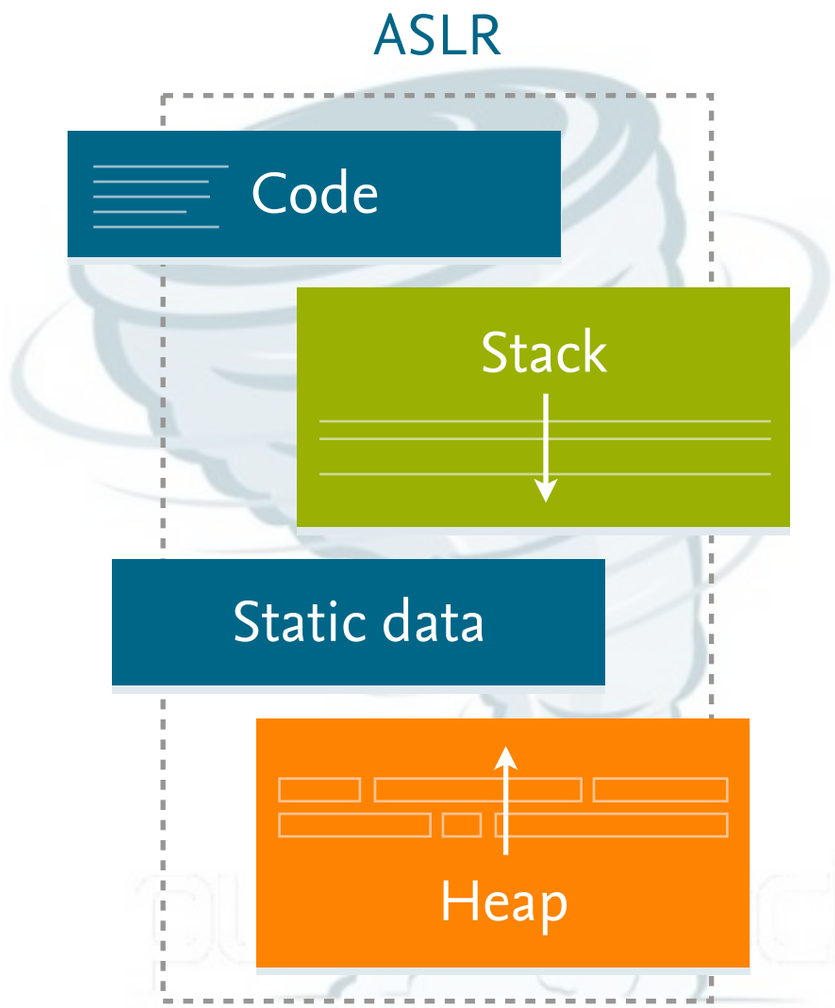
Non-Executable Memory

- **Access control on memory segments**
  - Writable (W) or executable (X)
  - Mutual exclusive use of W and X
  - Different implementations:
    DEP, PAX, W^X, ...
- **Reactive defense at run-time**
  - Overflow still possible
  - However: buffer non-executable
- **Hardens execution of injected code**



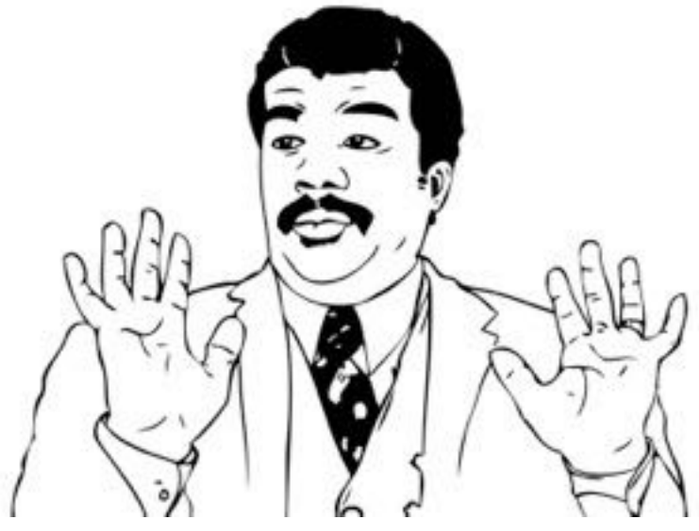
Randomization

- **Randomization of memory layout**
  - Shuffling of data in memory
  - Different granularity levels
- **Reactive defense at run-time**
  - Overflows still possible
  - However: addresses not predictable
- **Hardens exploitation of memory corruption vulnerabilities**



Constant Arms Race

- **What's the problem?**
  - Inconsistent abstraction of semantics → insecure functions
  - Mixing of control structures and data → heap and stack
  - Basic problem of designing secure software
- **Impressive creativity of attackers**
  - Return-oriented programming
  - Heap spraying
  - Rowhammer attacks
  - Spectre and MeltDown exploits



Summary

Summary

- **Vulnerabilities and exploits**
  - Vulnerabilities key problem in computer security
  - Various causes and types of vulnerabilities
  - Examples: Exploitation of heap and stack overflows
- **Defenses**
  - Mainly hardening of exploitation → fixing symptoms
  - Boundary checks, access control, randomization
  - On-going arms race with exploit authors