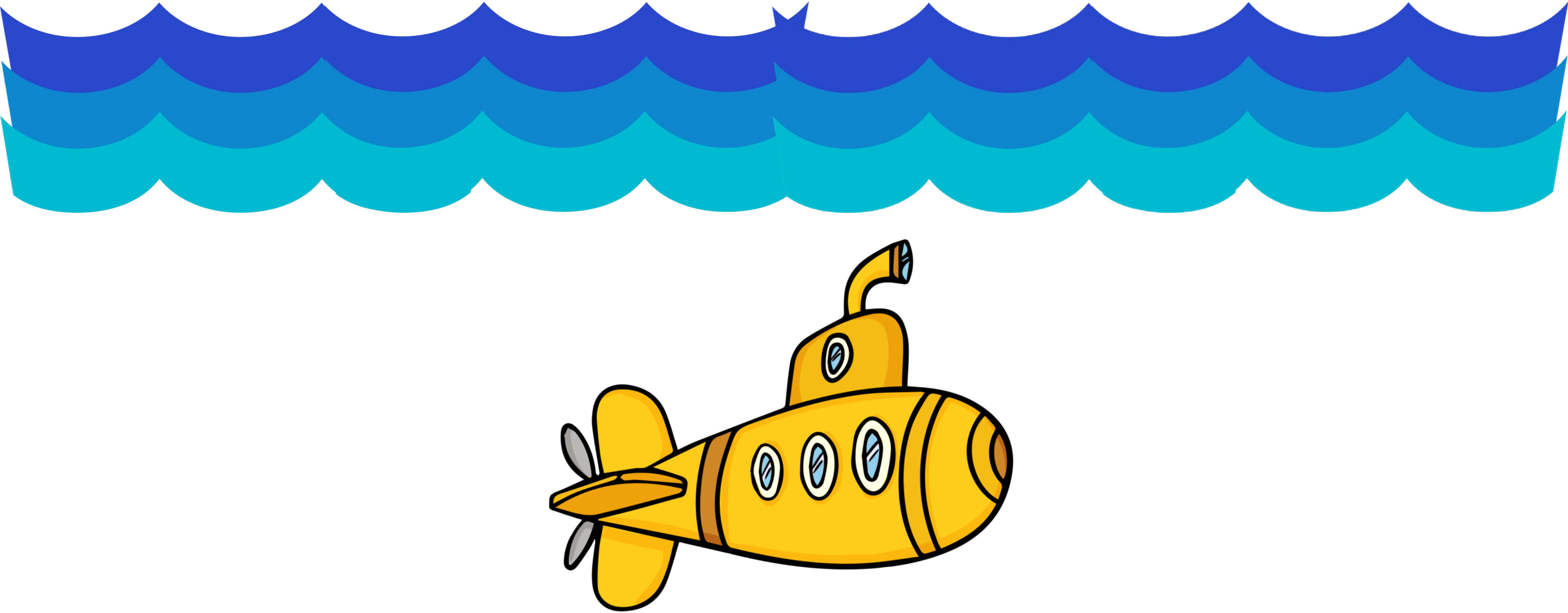
A Story of Under-C Discovery and Adventure

A look at Memory Safety

Miguel A. Arroyo

@miguelaarroyo12

The Evolution of Memory Safety



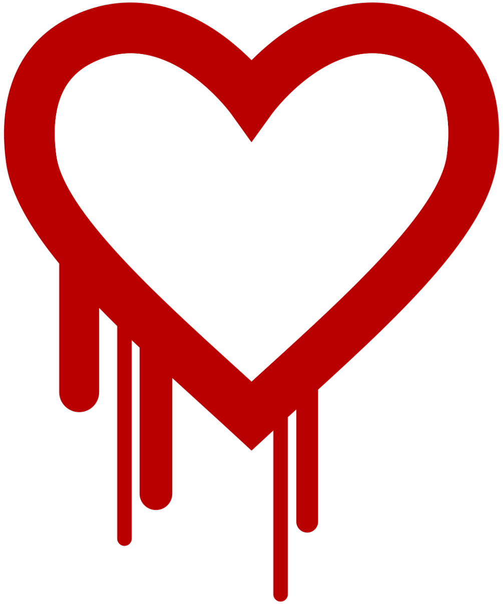
The Evolution of Memory Safety

The Morris Worm (1988)

Reference: Hilarie Orman - The Morris Worm: A Fifteen-Year Perspective

The Evolution of Memory Safety

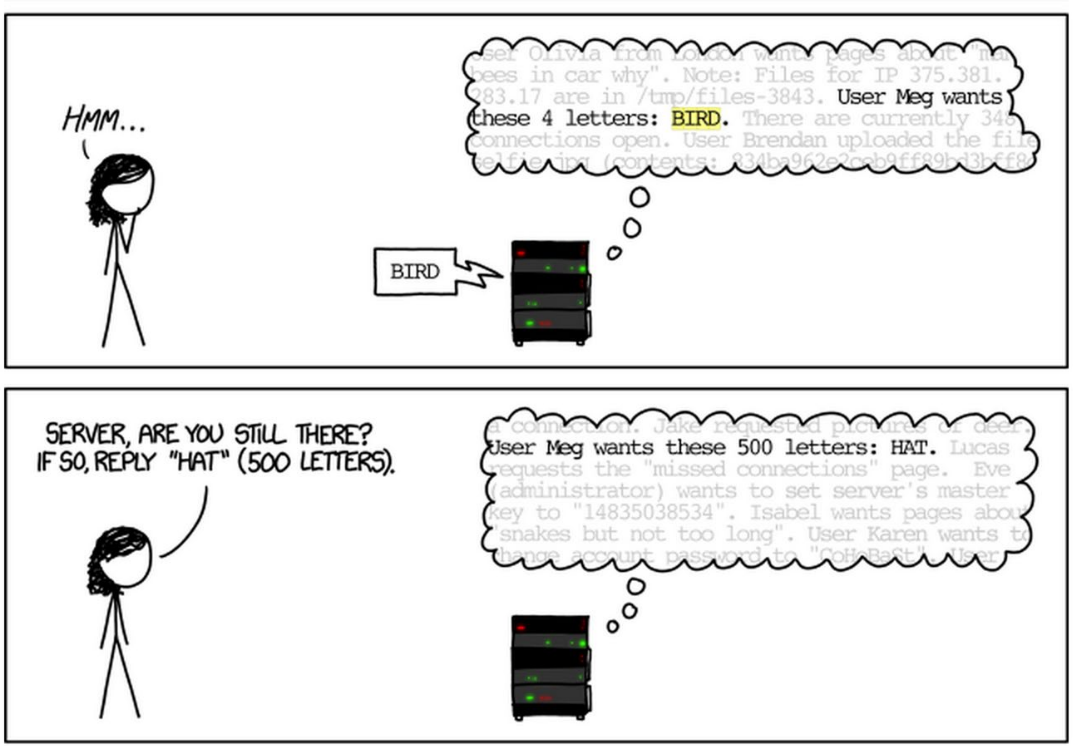
Heartbleed (2014)



Reference: Durumeric et al. - The Matter of Heartbleed

The Evolution of Memory Safety

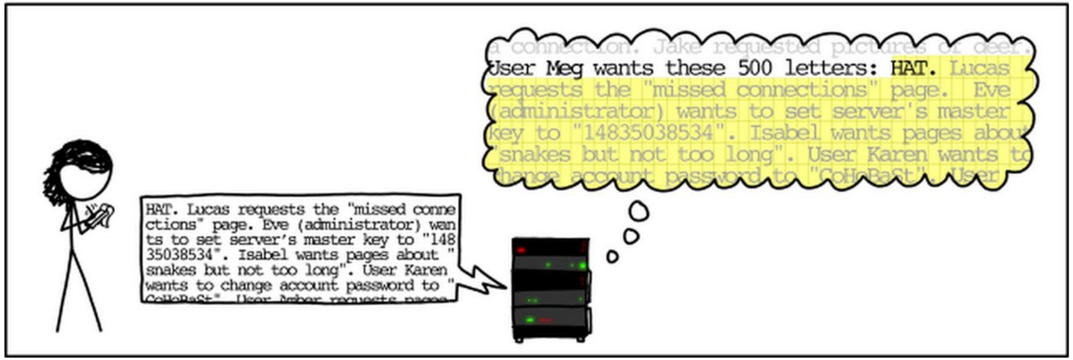
Heartbleed (2014)


HMM...
User Olivia from London wants pages about "man bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 348 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User

Source: https://xkcd.com/1354/

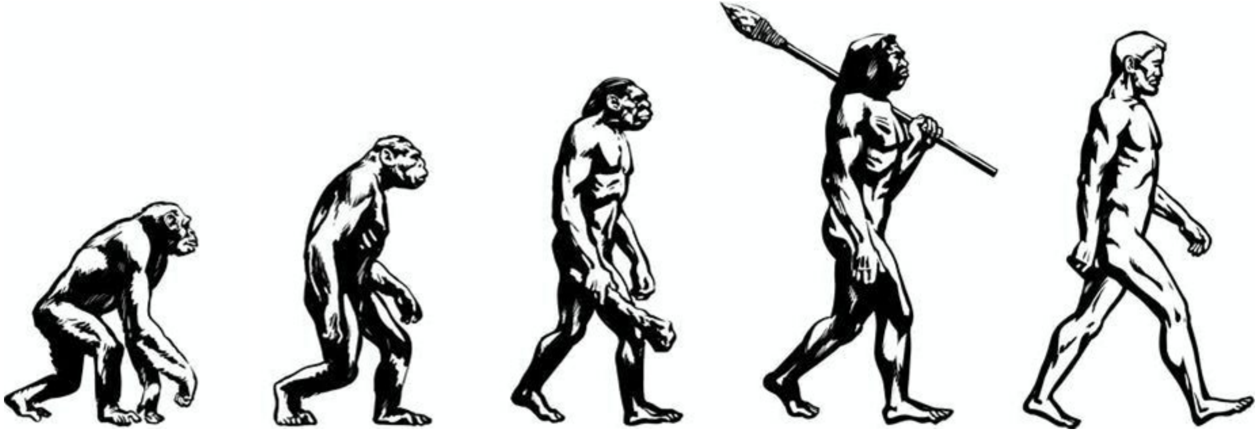
The Evolution of Memory Safety

Heartbleed (2014)

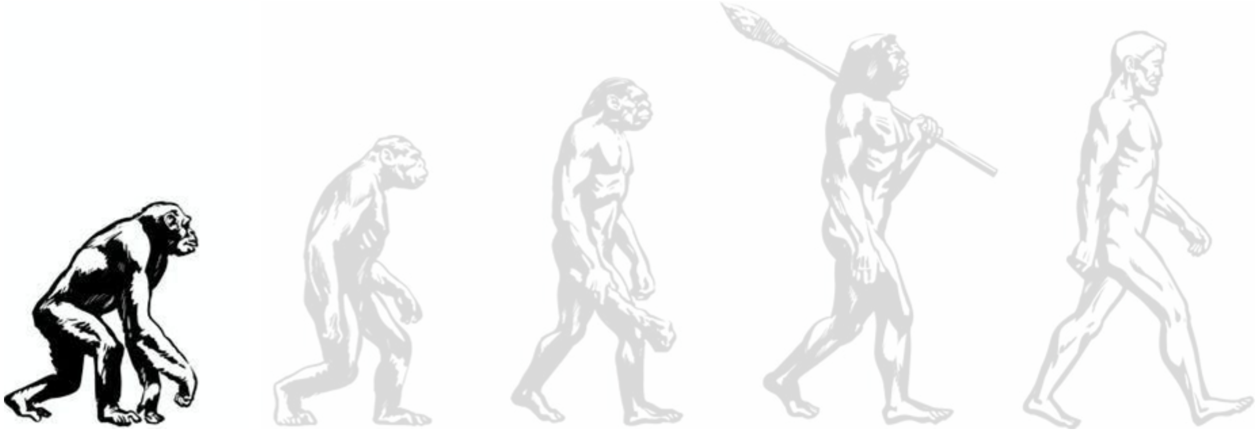

a connection. Jake requested pictures of deer.
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User Amber requests pages

Source: https://xkcd.com/1354/

The Evolution of Memory Safety

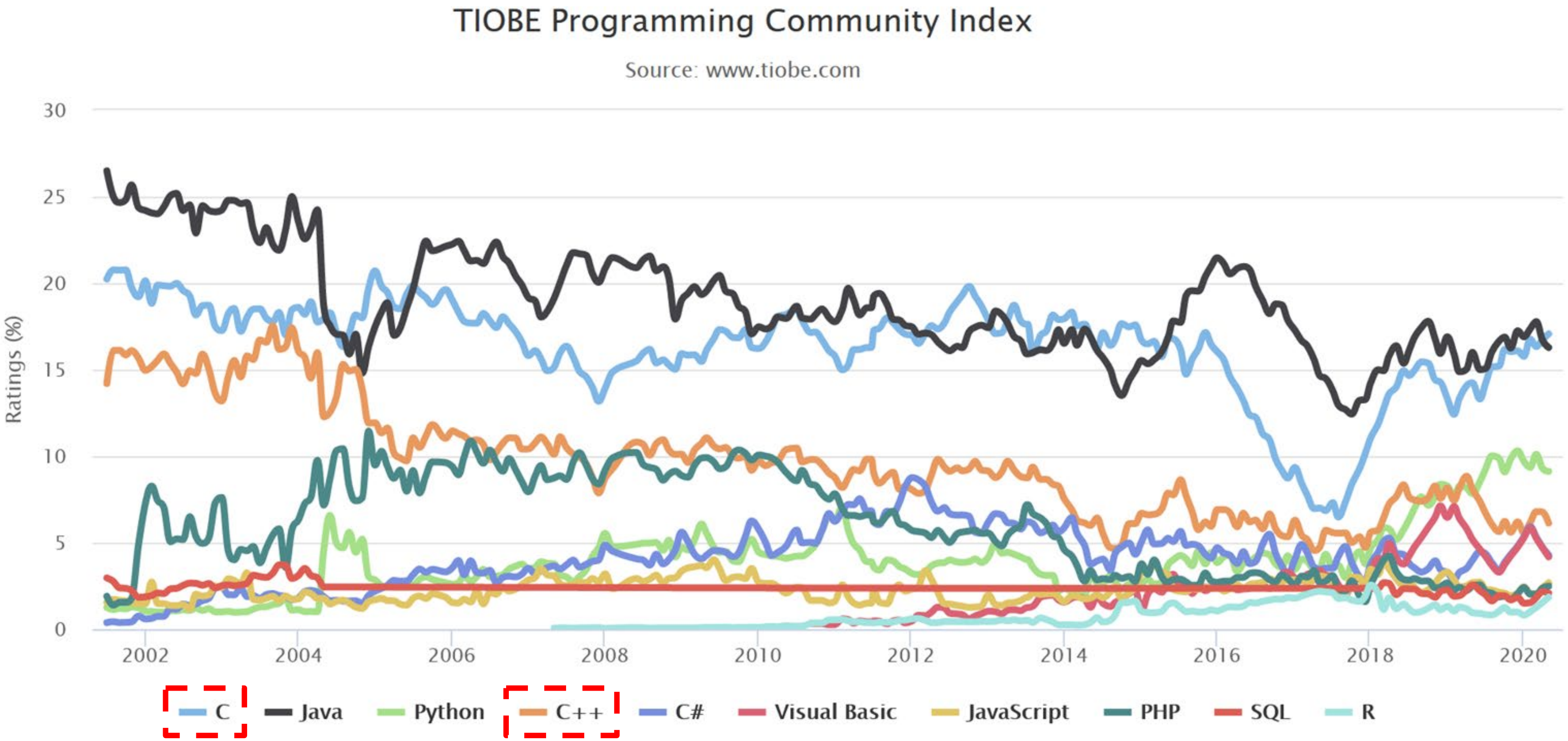


The Evolution of Memory Safety

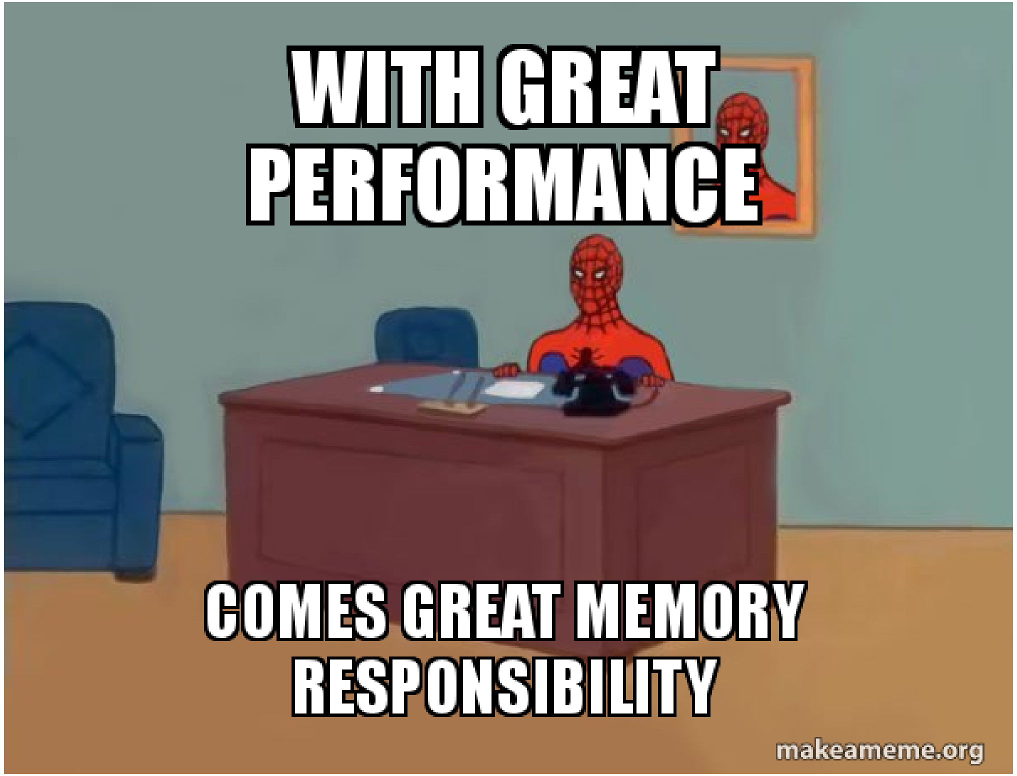


The fundamental vulnerabilities have remained the same!

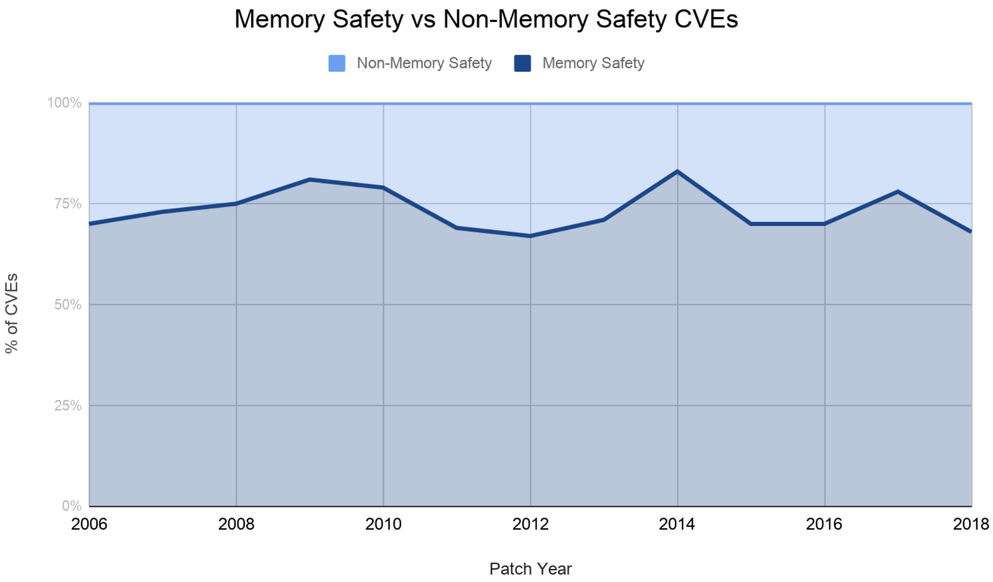
Software is Unsafe



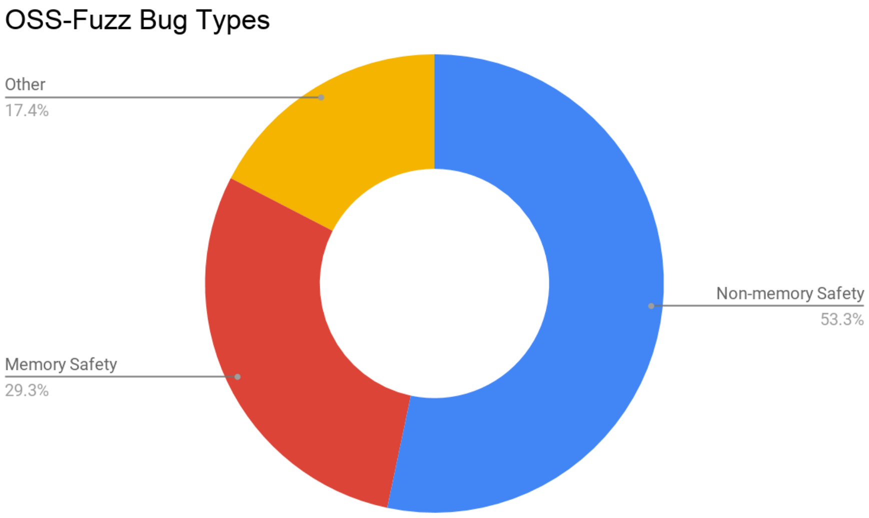
Software is Unsafe



Prevalence of Memory Safety Vulns



Microsoft Product CVEs



Google OSS-Fuzz bugs from 2016-2018.

Ref: Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019

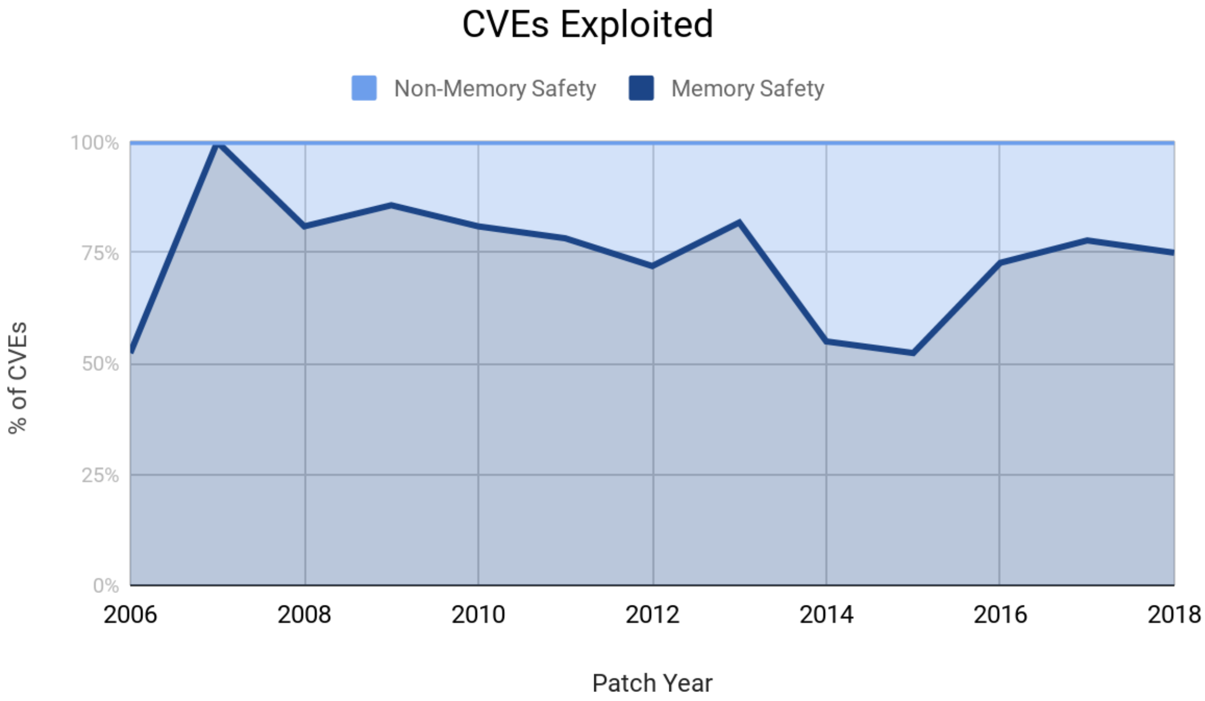
Ref: https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html

ATTACKERS

♥

MEMORY SAFETY

Attackers Prefer Memory Safety Vulns



Microsoft Product Exploits

Reference: Matt Miller, Microsoft Security Response Center (MSRC) - BlueHat 2019

What is Memory Safety?

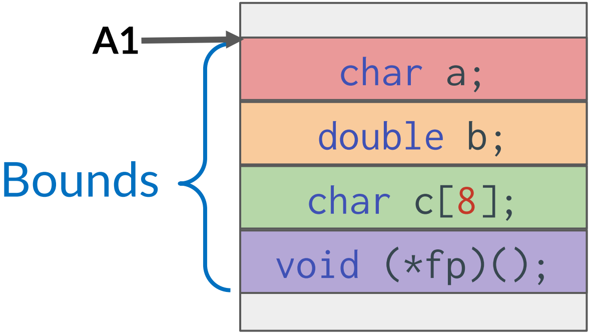
What is Memory Safety?

```
typedef struct {
    char a;
    double b;
    char c[8];
    void (*fp)();
} A_t;
```

```
A_t *A1 = malloc(
     sizeof(A_t));

free(A1);


A_t *A2 = malloc(

     sizeof(A_t));
```

Reference: https://hacks.mozilla.org/2019/01/fearless-security-memory-safety/

What is Memory Safety?



A1

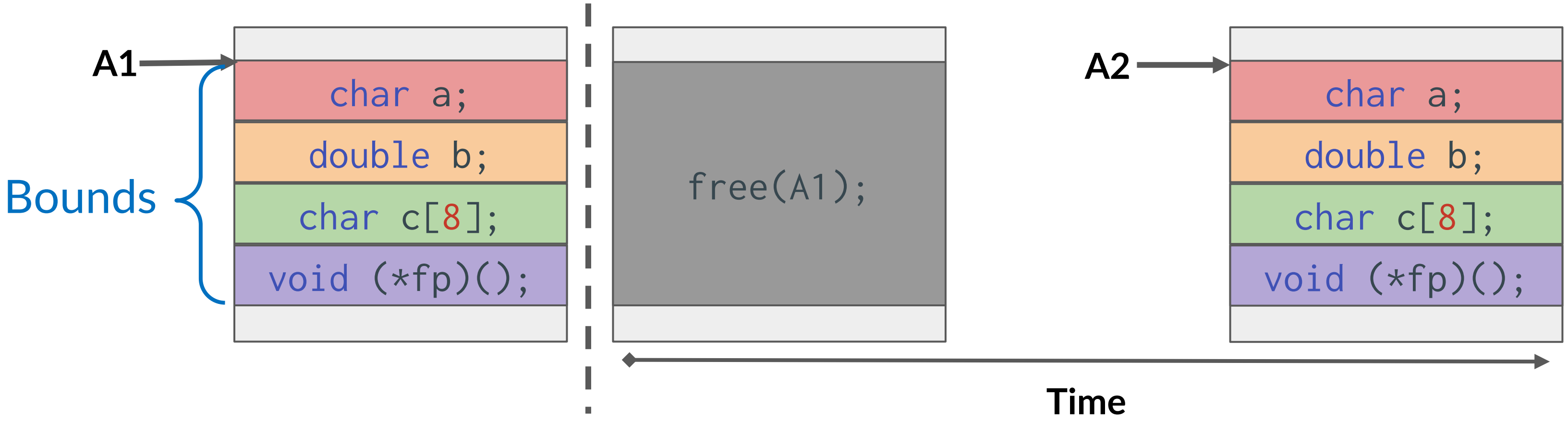
Bounds

```
char a;
double b;
char c[8];
void (*fp)();
```

(1) Spatial

eg. Overflows

What is Memory Safety?



A1

Bounds

```
char a;
double b;
char c[8];
void (*fp)();
```

```
free(A1);
```

A2

```
char a;
double b;
char c[8];
void (*fp)();
```

Time

(1) Spatial

eg. Overflows

(2) Temporal

eg. use-after-free

Why is Memory Safety still a problem?

Why is Memory Safety still a problem?

Defenses suffer from

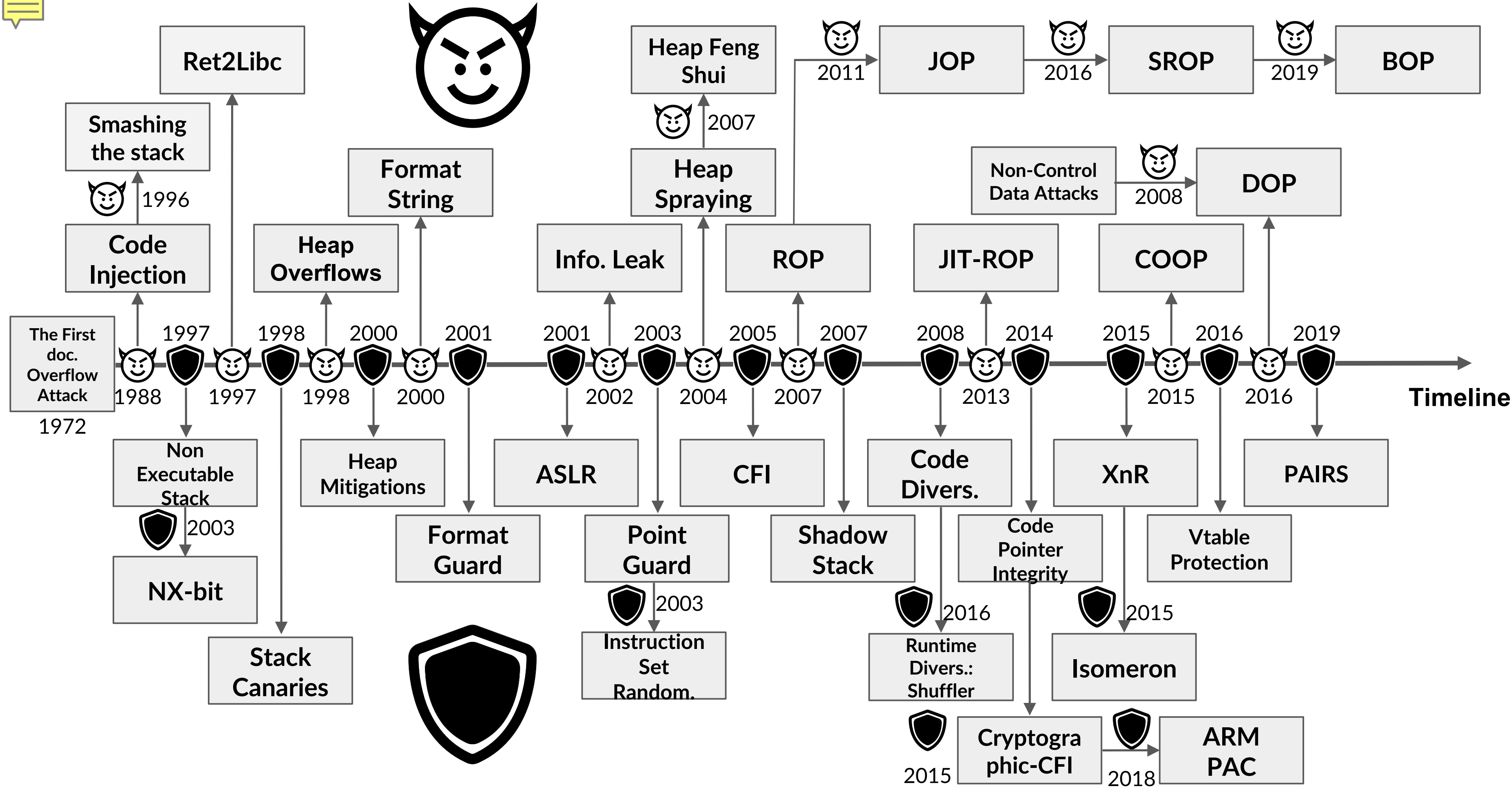
Performance Overheads

Costly Implementation

Compatibility

Reference: WarGames in memory: shall we play a game?

The Security Cat & Mouse Game



The First doc. Overflow Attack
1972

1988
Code Injection
1996
Smashing the stack

1997
Non Executable Stack
2003
NX-bit

1997
Ret2Libc

1998
Stack Canaries

1998
Heap Overflows

2000
Heap Mitigations

2000
Format String

2001
Format Guard

2001
ASLR

2002
Info. Leak

2003
Point Guard
2003
Instruction Set Random.

2004
Heap Spraying
2007
Heap Feng Shui

2005
CFI

2007
ROP
2011
JOP
2016
SROP
2019
BOP

2007
Shadow Stack

2008
Code Divers.
2016
Runtime Divers.: Shuffler

2013
JIT-ROP

2014
Code Pointer Integrity
2015
Cryptographic-CFI
2018
ARM PAC

Non-Control Data Attacks
2008
DOP

2015
XnR
2015
Isomeron

2015
COOP

2016
Vtable Protection

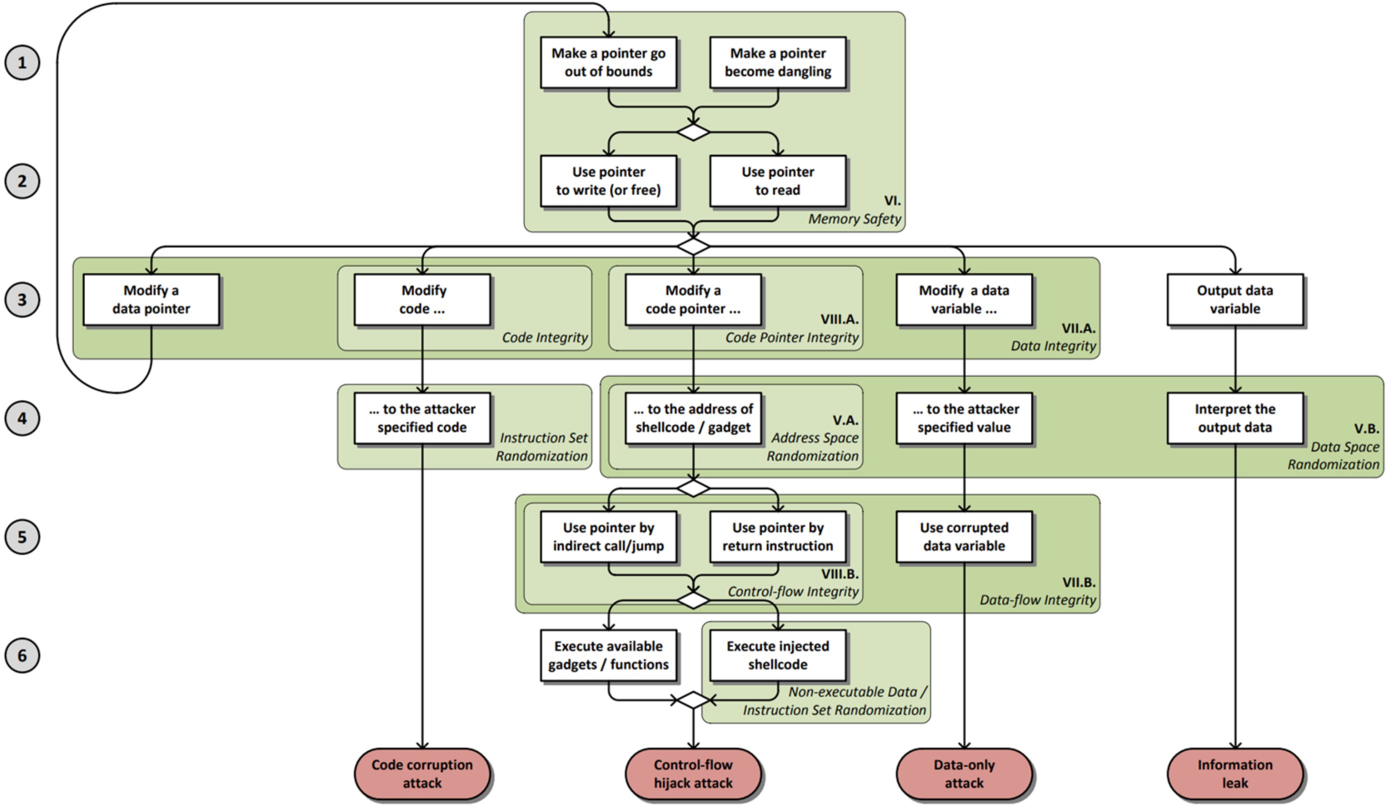
2016
DOP

2019
PAIRS

Timeline

Reference:Mohamed Tarek Ibn Ziad @ shorturl.at/muJKO

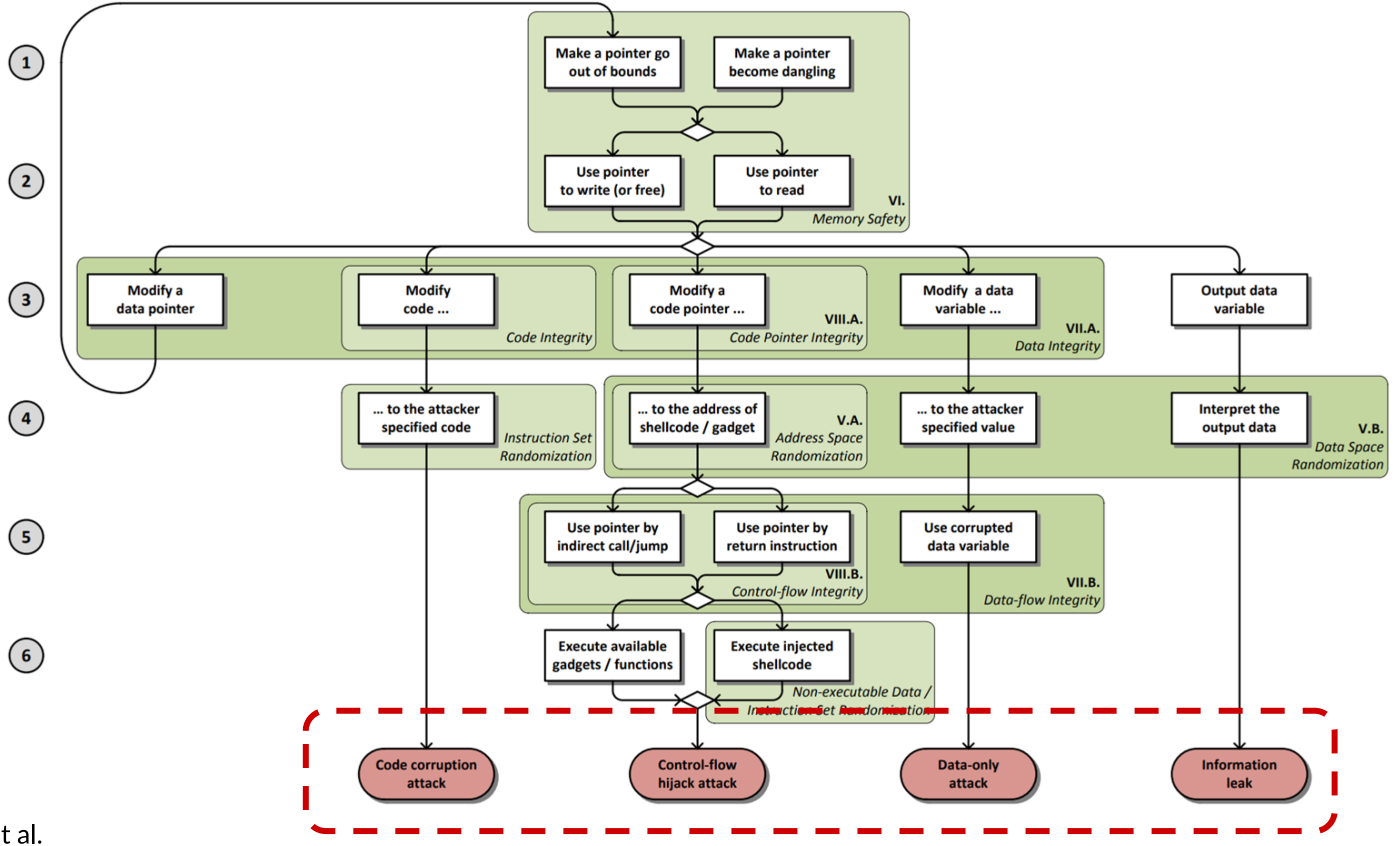
The Memory Attack Model



1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write (or free)
Use pointer to read
VI.
Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
VIII.A.
Code Pointer Integrity
Modify a data variable ...
VII.A.
Data Integrity
Output data variable
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
V.A.
Address Space Randomization
... to the attacker specified value
Interpret the output data
V.B.
Data Space Randomization
Use pointer by indirect call/jump
Use pointer by return instruction
VIII.B.
Control-flow Integrity
Use corrupted data variable
VII.B.
Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data / Instruction Set Randomization
Code corruption attack
Control-flow hijack attack
Data-only attack
Information leak

Reference: Szekeres et al.
SoK: Eternal War in Memory

The Memory Attack Model



1

2

3

4

5

6

Make a pointer go out of bounds

Make a pointer become dangling

Use pointer to write (or free)

Use pointer to read

VI.
Memory Safety

Modify a data pointer

Modify code ...

Code Integrity

Modify a code pointer ...

VIII.A.
Code Pointer Integrity

Modify a data variable ...

VII.A.
Data Integrity

Output data variable

... to the attacker specified code

Instruction Set Randomization

... to the address of shellcode / gadget

V.A.
Address Space Randomization

... to the attacker specified value

Interpret the output data

V.B.
Data Space Randomization

Use pointer by indirect call/jump

Use pointer by return instruction

VIII.B.
Control-flow Integrity

Use corrupted data variable

VII.B.
Data-flow Integrity

Execute available gadgets / functions

Execute injected shellcode

Non-executable Data / Instruction Set Randomization

Code corruption attack

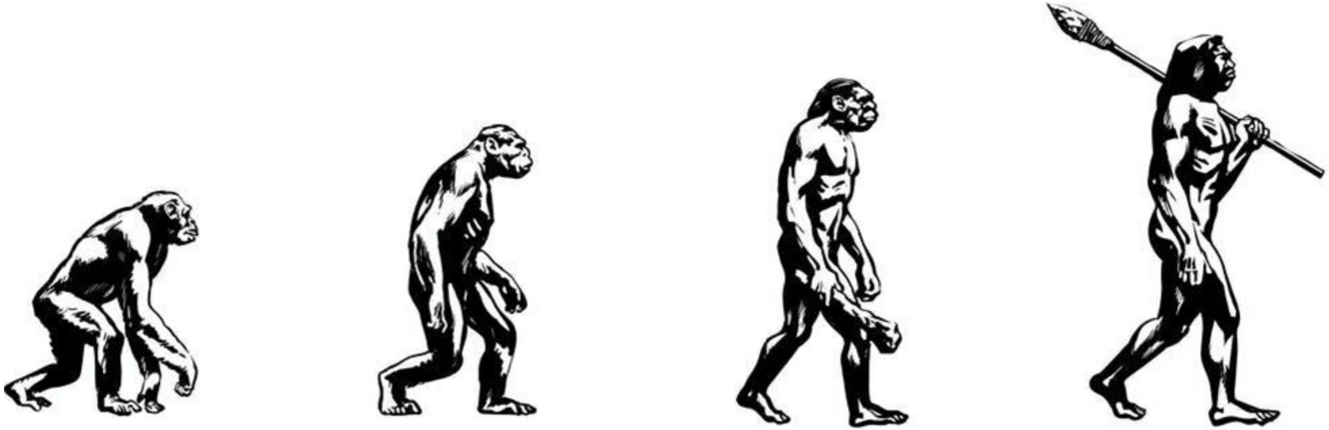
Control-flow hijack attack

Data-only attack

Information leak

Reference: Szekeres et al.
SoK: Eternal War in Memory

The Memory Attack Model



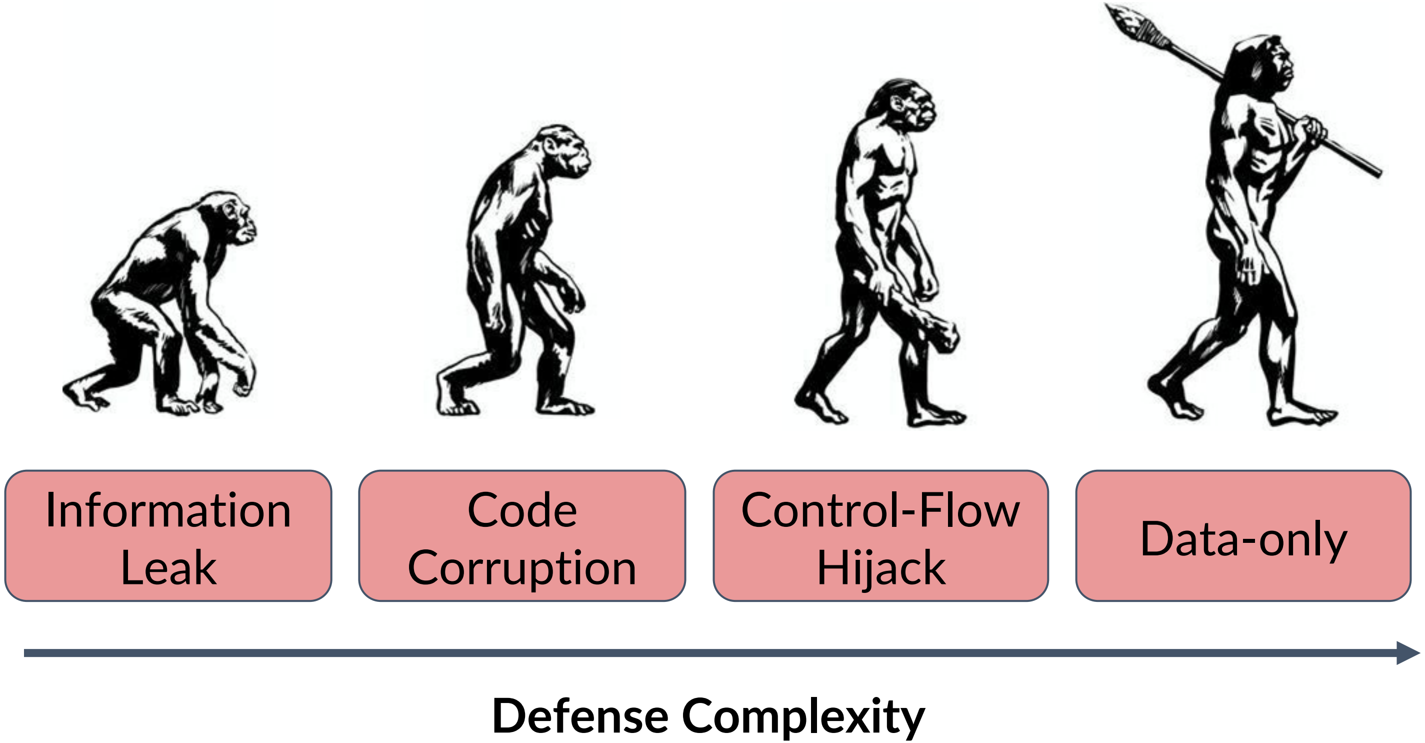
Information Leak

Code Corruption

Control-Flow Hijack

Data-only

The Memory Attack Model



Information Leak

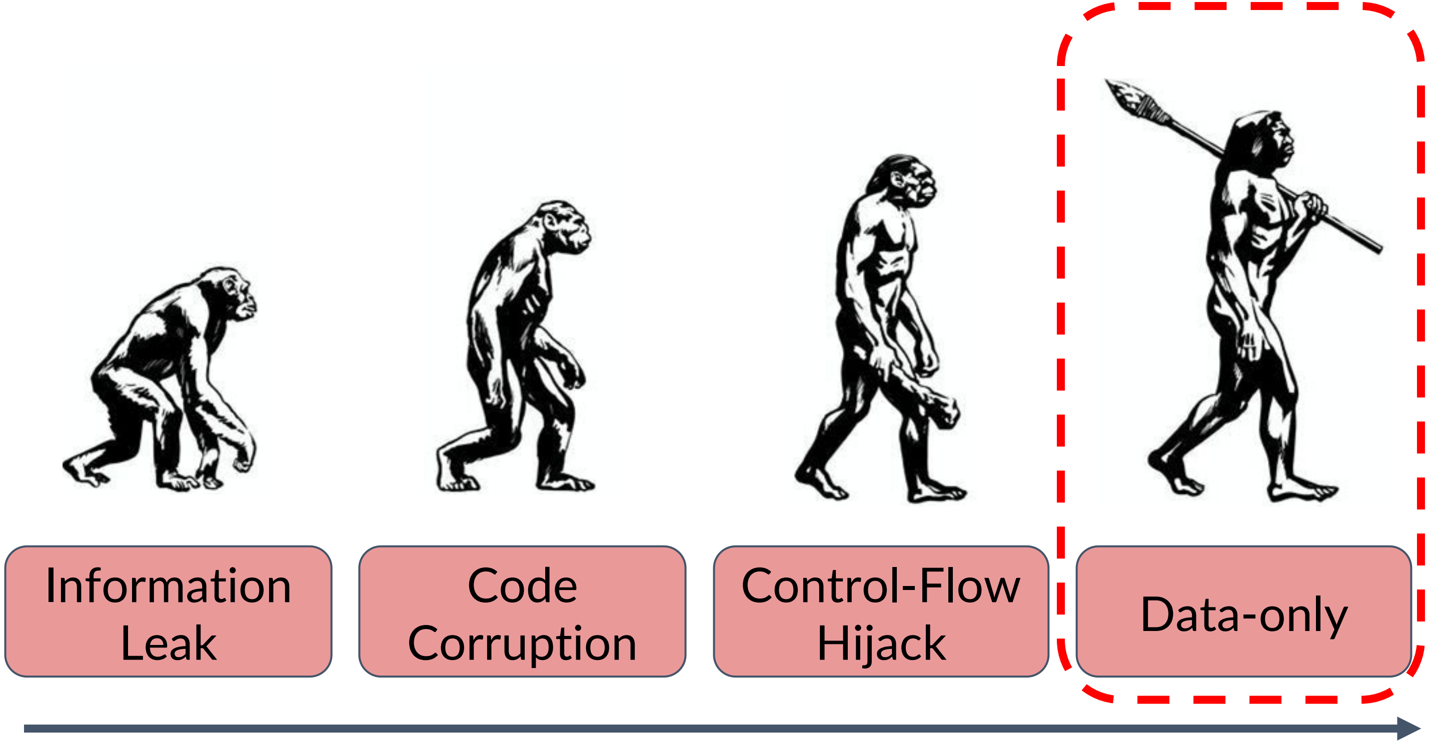
Code Corruption

Control-Flow Hijack

Data-only

Defense Complexity

The Memory Attack Model



Information Leak

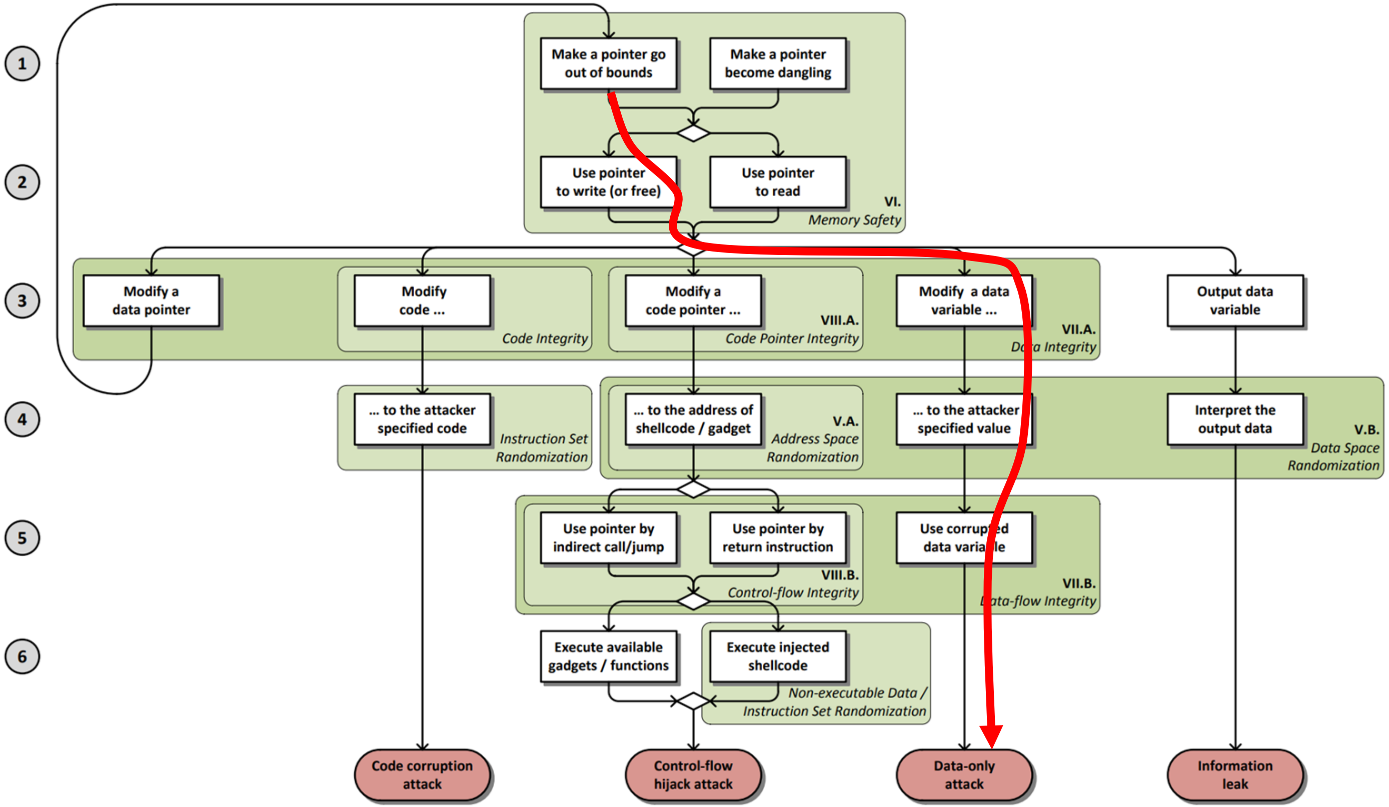
Code Corruption

Control-Flow Hijack

Data-only

Defense Complexity

The Memory Attack Model



1
2
3
4
5
6

Make a pointer go out of bounds

Make a pointer become dangling

Use pointer to write (or free)

Use pointer to read

VI. *Memory Safety*

Modify a data pointer

Modify code ...

Code Integrity

Modify a code pointer ...

VIII.A. *Code Pointer Integrity*

Modify a data variable ...

VII.A. *Data Integrity*

Output data variable

... to the attacker specified code

Instruction Set Randomization

... to the address of shellcode / gadget

V.A. *Address Space Randomization*

... to the attacker specified value

Interpret the output data

V.B. *Data Space Randomization*

Use pointer by indirect call/jump

Use pointer by return instruction

VIII.B. *Control-flow Integrity*

Use corrupted data variable

VII.B. *Data-flow Integrity*

Execute available gadgets / functions

Execute injected shellcode

Non-executable Data / Instruction Set Randomization

Code corruption attack

Control-flow hijack attack

Data-only attack

Information leak

Reference: Szekeres et al.
SoK: Eternal War in Memory

Data-only Attacks

State-of-the-art Exploit Techniques

Data-only Attacks

Direct Data Manipulation

Non-Control-Data Attacks Are Realistic Threats
Chen et al. (2005)

- An attacker directly manipulates the target data to accomplish the malicious goal.

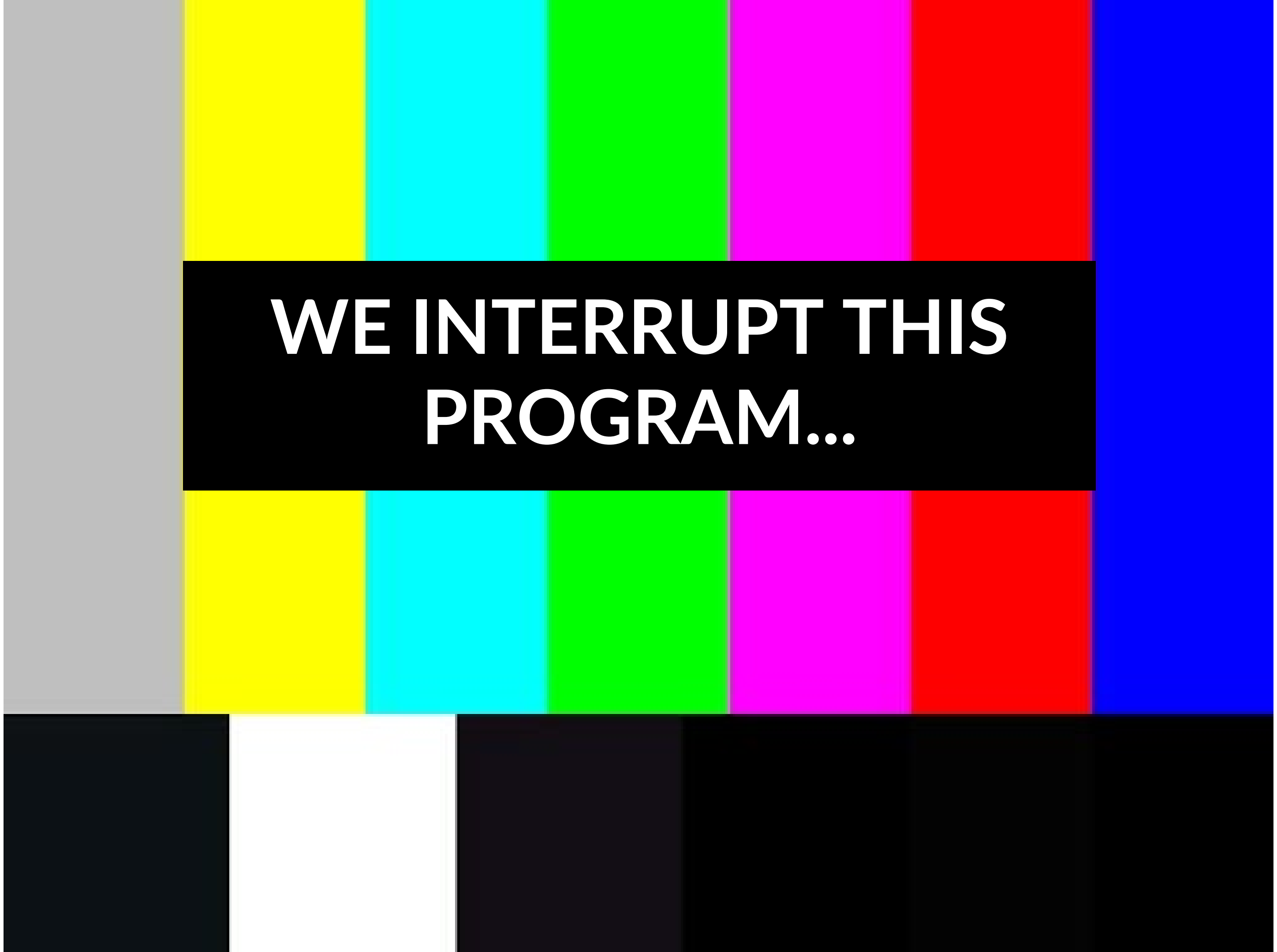
```
void foo(...) {
  ...
  bool is_admin = false;
  ...
  // Corrupt authenticated
  type = packet_read();
  ...
  if (is_admin) {
  // do privileged ops
  ...
  }
  ...
}
```

Data-only Attacks

Data-Oriented Programming (DOP)

Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks
Hu et al. (2016)

- An attacker performs arbitrary computations in program memory by chaining the execution of short sequences of instructions (referred to as *gadgets*).

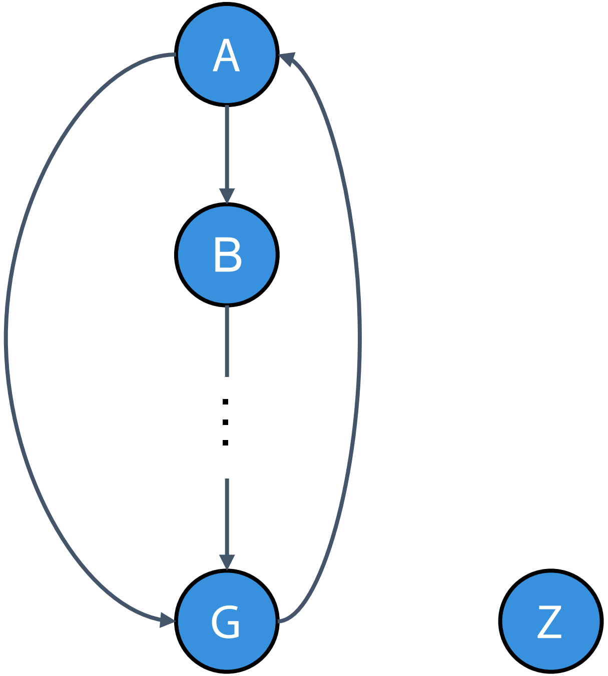


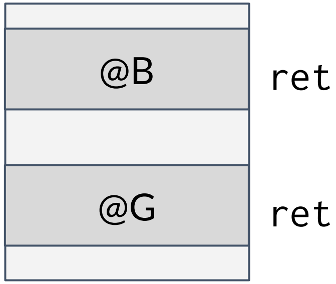
WE INTERRUPT THIS PROGRAM…

Review

Return-Oriented Programming (ROP)

https://miguel.arroyo.me/resources



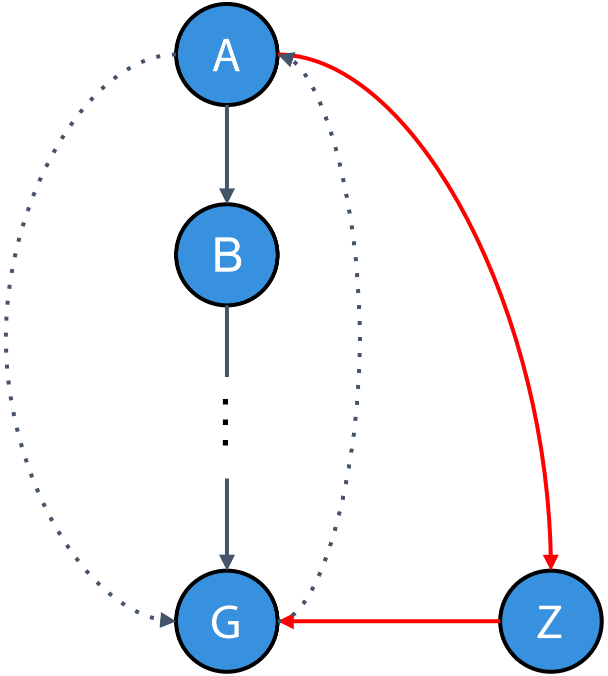


Stack

Review

Return-Oriented Programming (ROP)

https://miguel.arroyo.me/resources



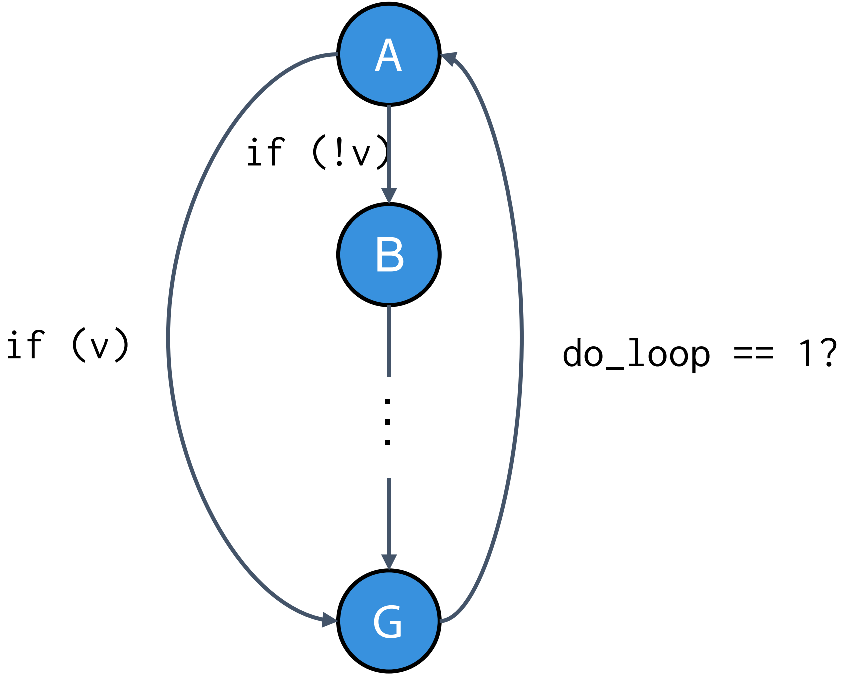
@Z	ret
@G	ret

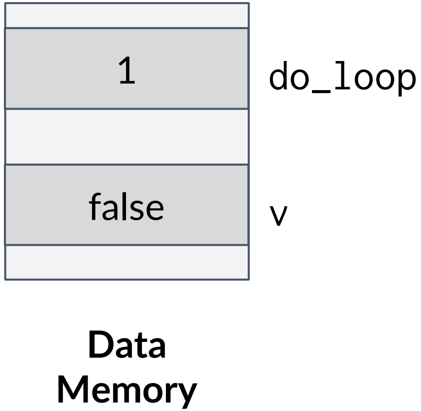
Stack

AND NOW BACK TO THE SCHEDULED PROGRAMMING

Data-only Attacks

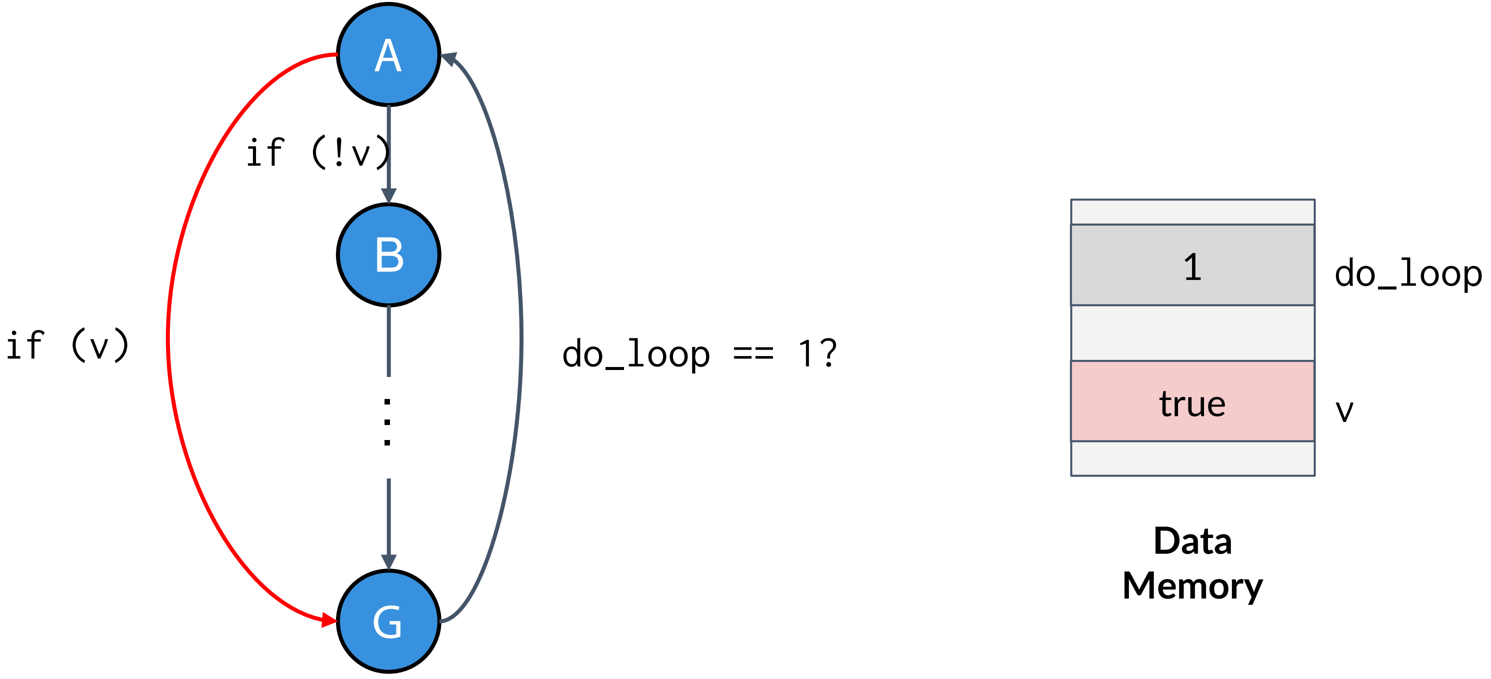
Data-Oriented Programming (DOP)





Data-only Attacks

Data-Oriented Programming (DOP)



Data-Oriented Programming

Motivating Example

```c
struct server{int *cur_max, total, typ;} *srv;
int quota = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12]
...
while (quota--) {
  readData(sockfd, buf); // stack bof
  if(*type == NONE ) break;
  if(*type == STREAM)
    *size = *(srv->cur_max);
  else {
    srv->typ = *type;
    srv->total += *size;
  } //...(following code skipped)...
}
```

```c
struct Obj {struct Obj *next; int prop;}

void updateList(struct Obj *list, int addend){
  for(; list != NULL; list = list->next)
    list->prop += addend;
}
```

Source: http://www.ieee-security.org/TC/SP2016/slides/25-4/hu.pdf

Data-Oriented Programming

Motivating Example

```
struct server{int *cur_max, total, typ;} *srv;
int quota = MAXCONN; int *size, *type;
char buf[MAXLEN];
size = &buf[8]; type = &buf[12]
...
while (quota--) {
  readData(sockfd, buf); // stack bof
  if(*type == NONE ) break;
  if(*type == STREAM)
    *size = *(srv->cur_max);
  else {
    srv->typ = *type;
    srv->total += *size;
  } //...(following code skipped)...
}
```

?

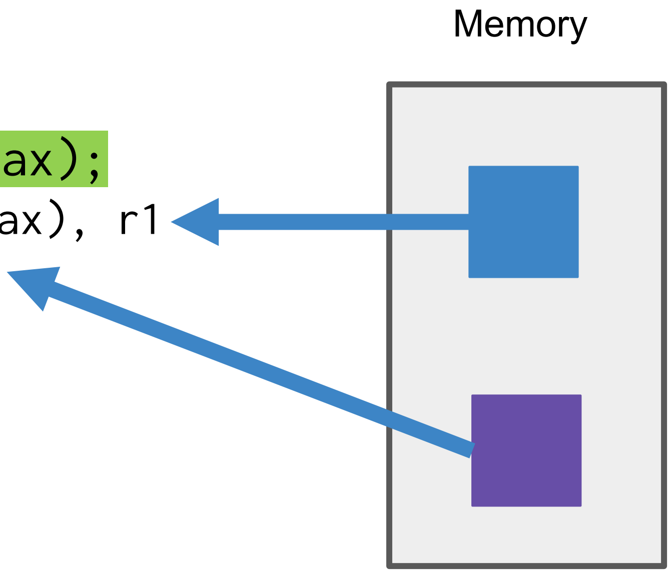
```
struct Obj {struct Obj *next; int prop;}

void updateList(struct Obj *list, int addend){
  for(; list != NULL; list = list->next)
    list->prop += addend;
}
```

Source: http://www.ieee-security.org/TC/SP2016/slides/25-4/hu.pdf

Data-Oriented Programming

DOP Gadgets

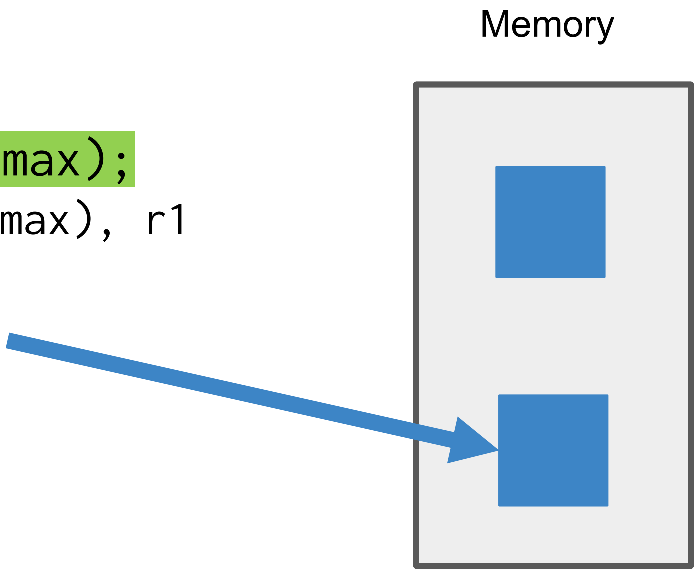
```
Load:*size = *(srv->cur_max);
        1.mov *(&srv->cur_max), r1
        2.mov *(&size), r2
        3.mov r1, *(&size)
```



Data-Oriented Programming

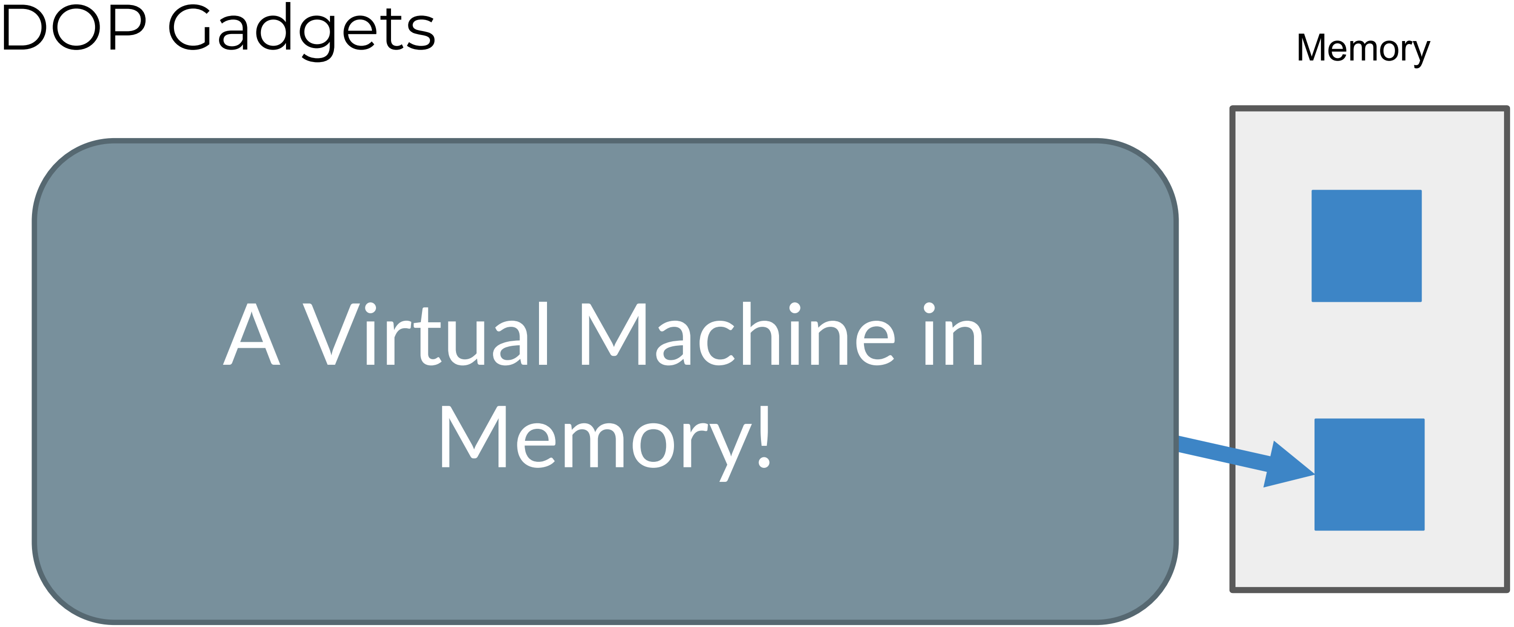
DOP Gadgets

```
Load:*size = *(srv->cur_max);
        1.mov *(&srv->cur_max), r1
        2.mov *(&size), r2
        3.mov r1, *(&size)
```



Data-Oriented Programming

DOP Gadgets



Motivating Example

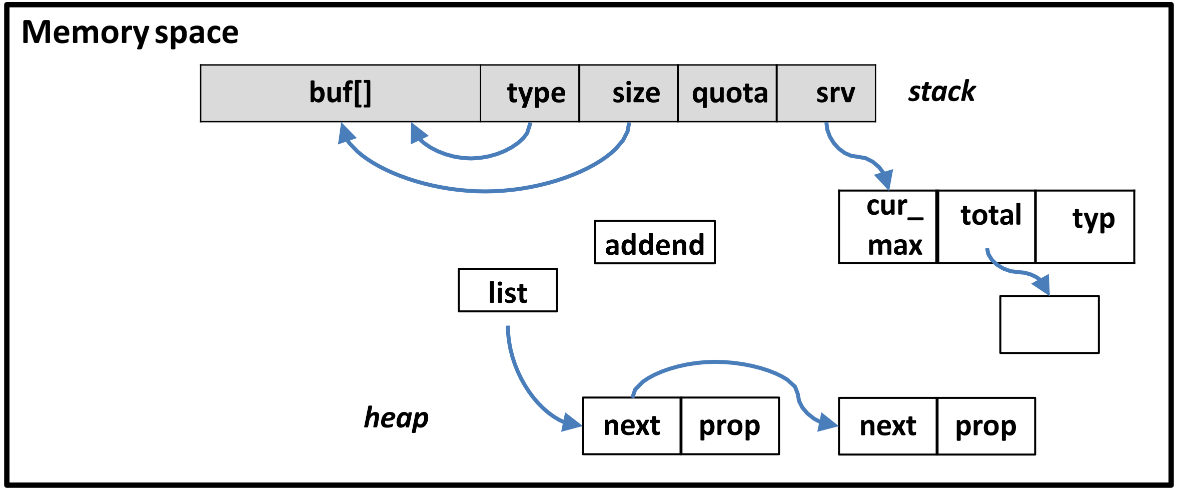
```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

vulnerable program

```
for(; list != NULL; list = list->next)
   list->prop += addend;
```

malicious computation

simulate ?



Memory space

buf[] | type | size | quota | srv

stack

addend

list

cur_max | total | typ

next | prop

next | prop

heap

Source: http://www.ieee-security.org/TC/SP2016/slides/25-4/hu.pdf

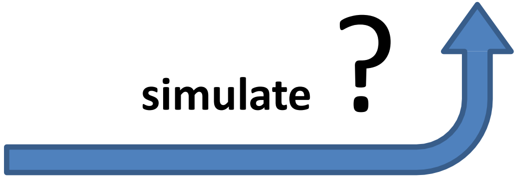
Motivating Example

```
while (quota--) {
    readData(sockfd, buf);
    if(*type == NONE ) break;
    if(*type == STREAM)
        *size = *(srv->cur_max);
    else {
        srv->typ = *type;
        srv->total += *size;
    }
}
```

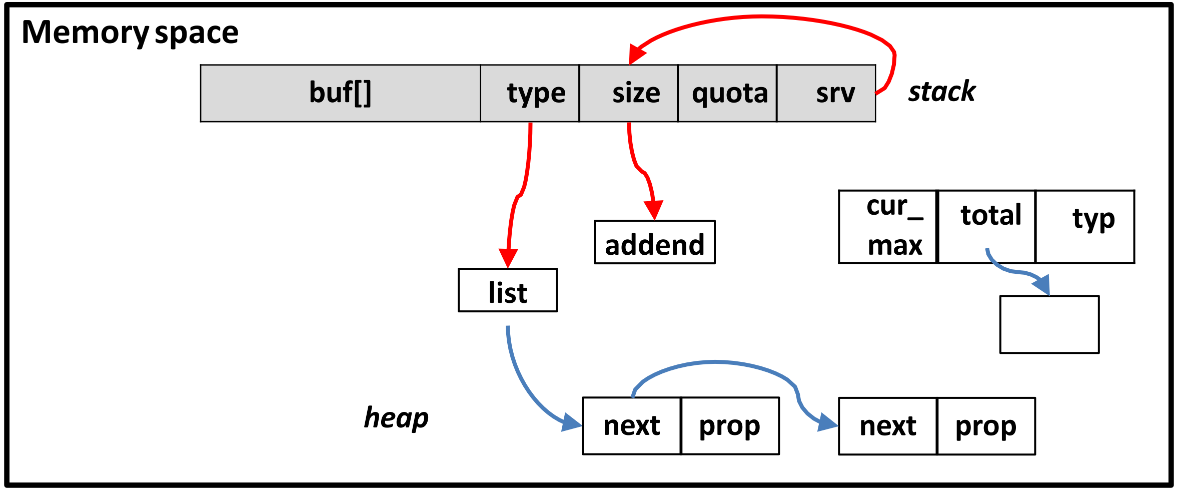
vulnerable program

```
for(; list != NULL; list = list->next)
    list->prop += addend;
```

malicious computation



simulate ?



Memory space

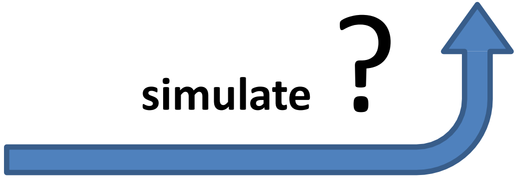
Motivating Example

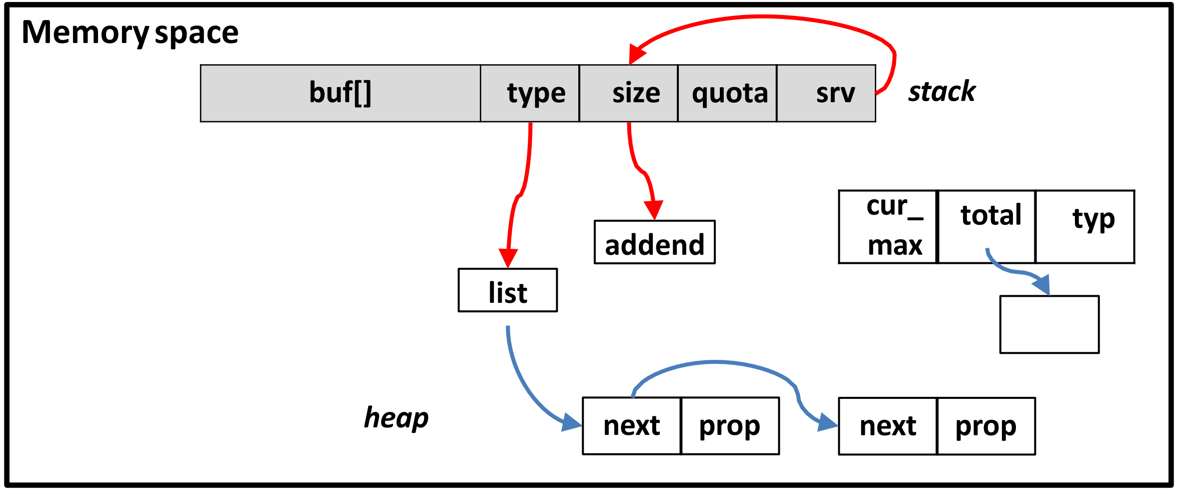
```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

vulnerable program

```
for(; list != NULL; list = list->next)
    list->prop += addend;
```

malicious computation





Motivating Example

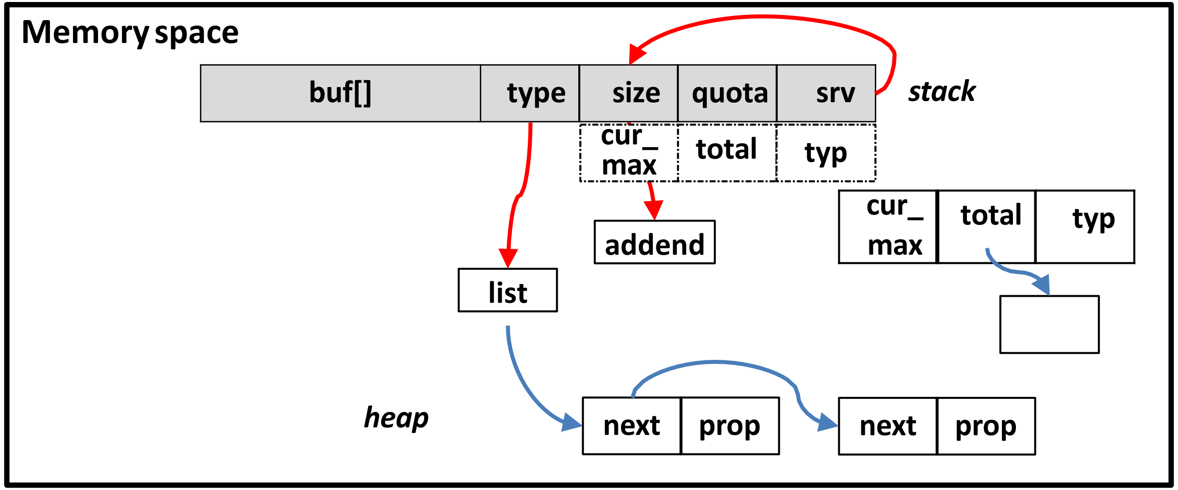
```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

vulnerable program

```
for(; list != NULL; list = list->next)
    list->prop += addend;
```

malicious computation

simulate ?



Motivating Example

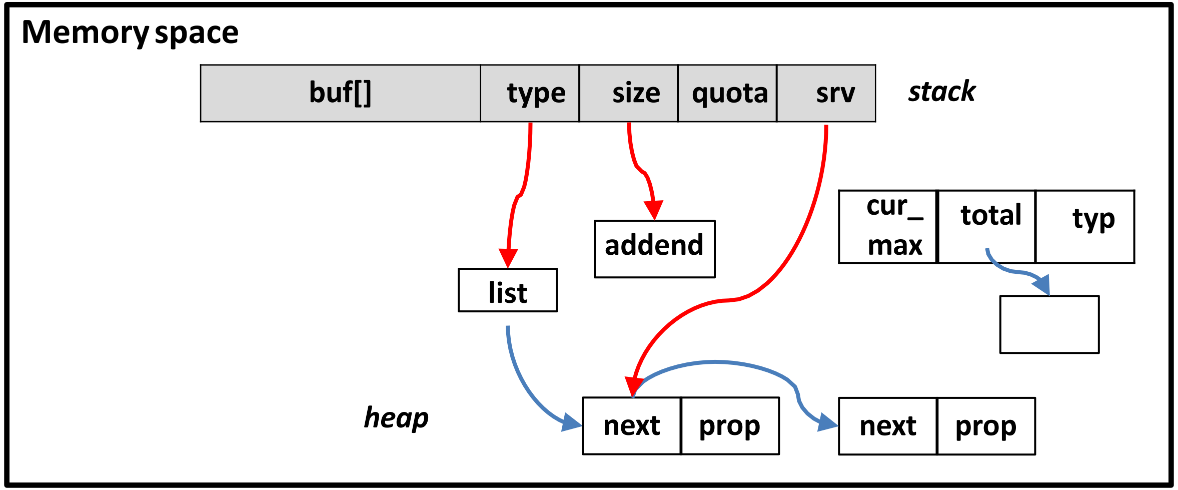
```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

vulnerable program

```
for(; list != NULL; list = list->next)
    list->prop += addend;
```

malicious computation

simulate ?



Motivating Example

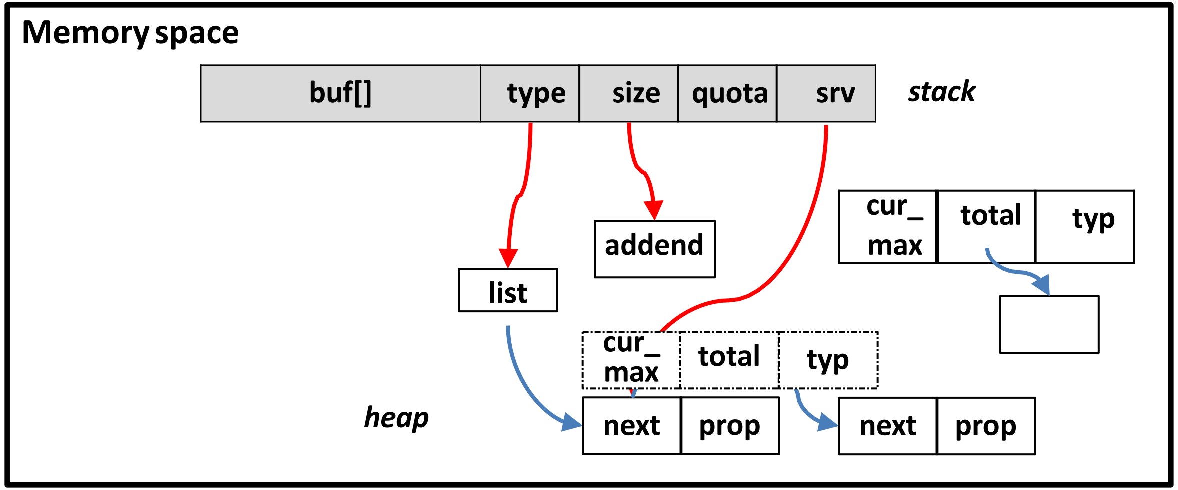
```
while (quota--) {
 readData(sockfd, buf);
 if(*type == NONE ) break;
 if(*type == STREAM)
     *size = *(srv->cur_max);
 else {
     srv->typ = *type;
     srv->total += *size;
 }
}
```

vulnerable program

```
for(; list != NULL; list = list->next)
  list->prop += addend;
```

malicious computation

simulate ?



Motivating Example

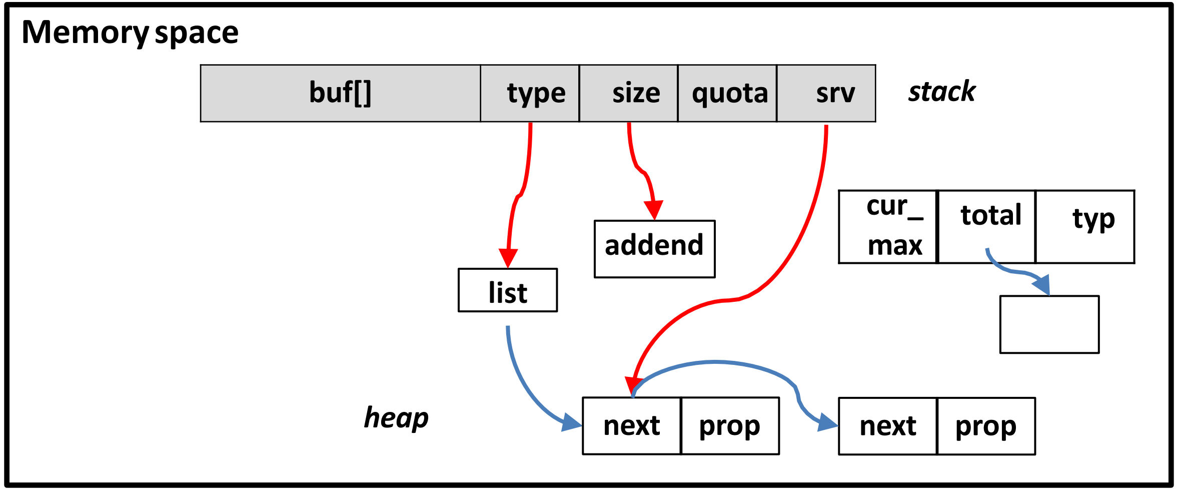
```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

vulnerable program

```
for(; list != NULL; list = list->next)
   list->prop += addend;
```

malicious computation

simulate ?



Motivating Example

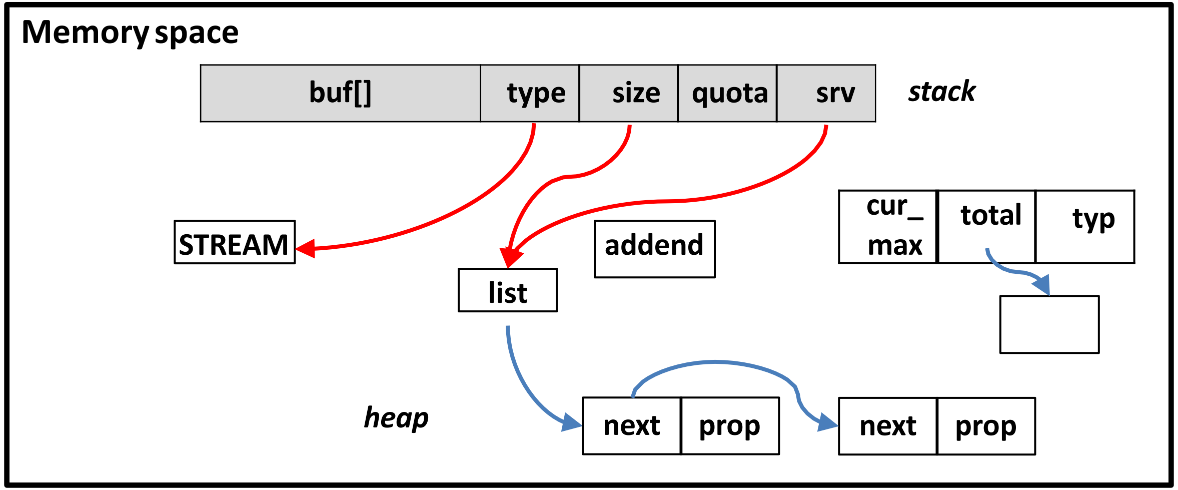
```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

vulnerable program

```
for(; list != NULL; list = list->next)
    list->prop += addend;
```

malicious computation

simulate ?



Motivating Example

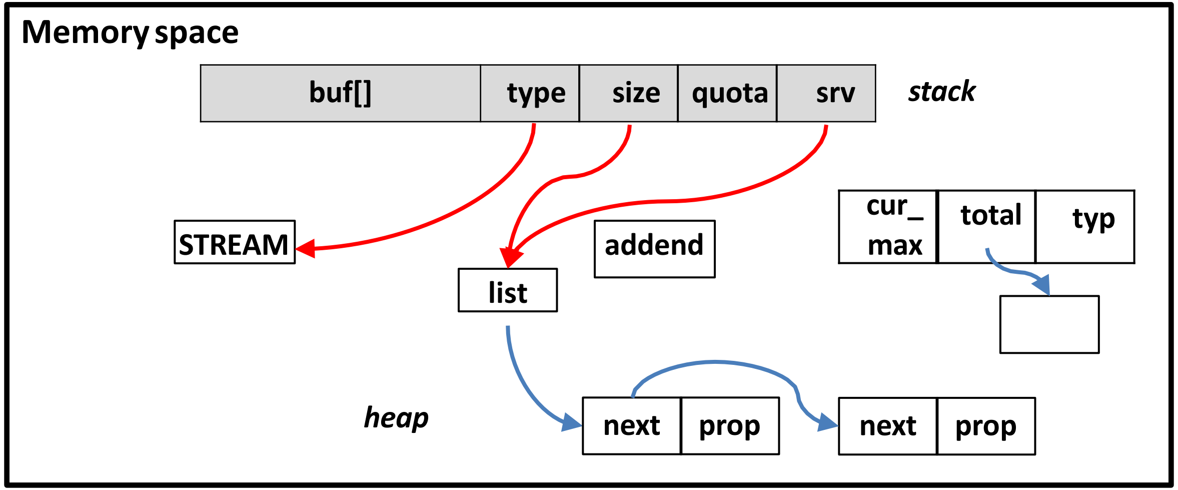
```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

vulnerable program

```
for(; list != NULL; list = list->next)
   list->prop += addend;
```

malicious computation

simulate ?



Motivating Example

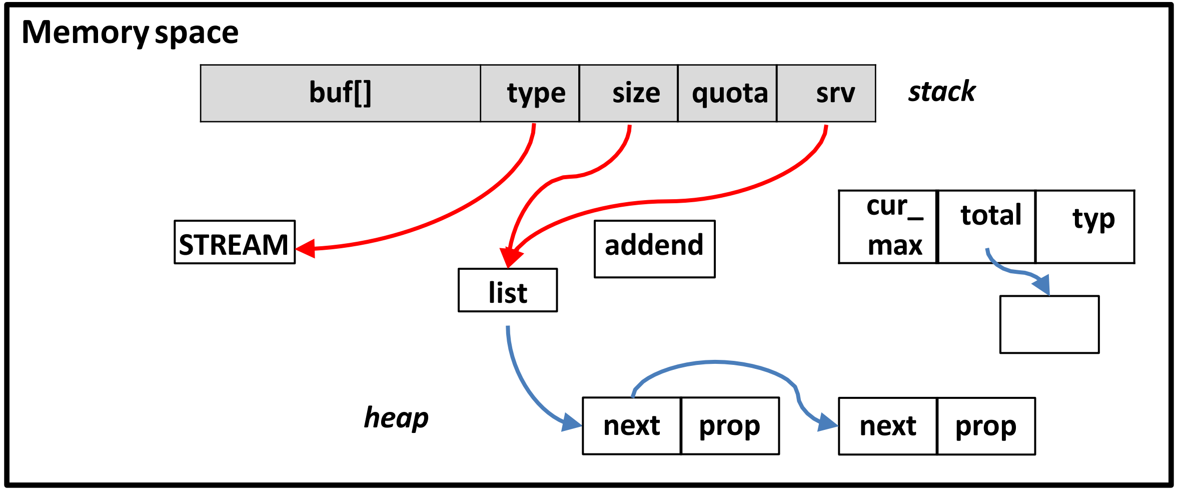
```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

vulnerable program

```
for(; list != NULL; list = list->next)
  list->prop += addend;
```

malicious computation

simulate ?



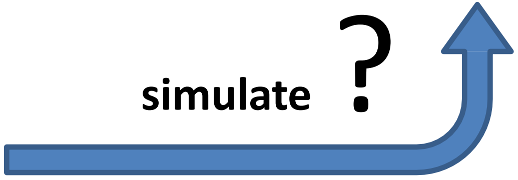
Motivating Example

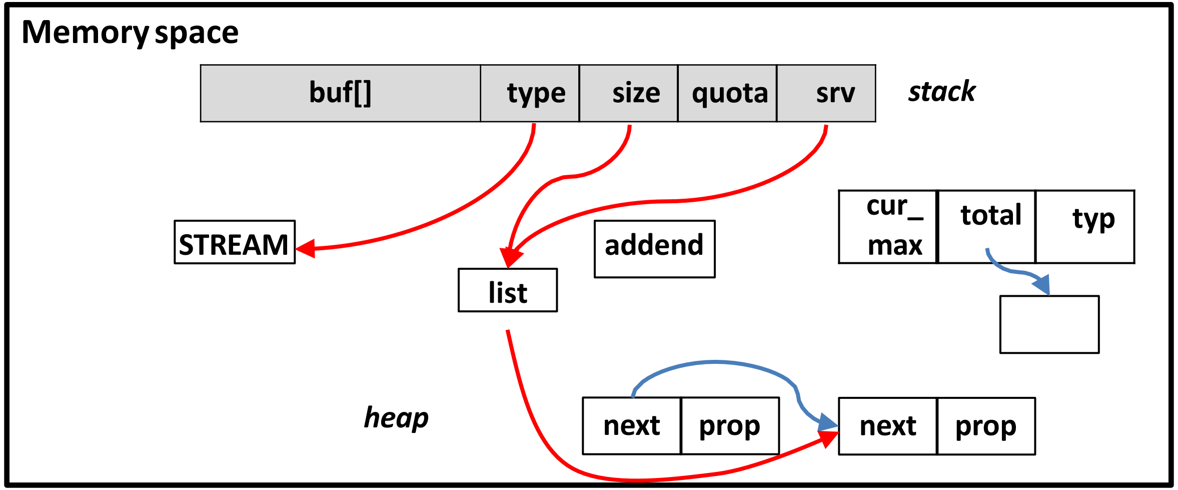
```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

vulnerable program

```
for(; list != NULL; list = list->next)
  list->prop += addend;
```

malicious computation





Motivating Example

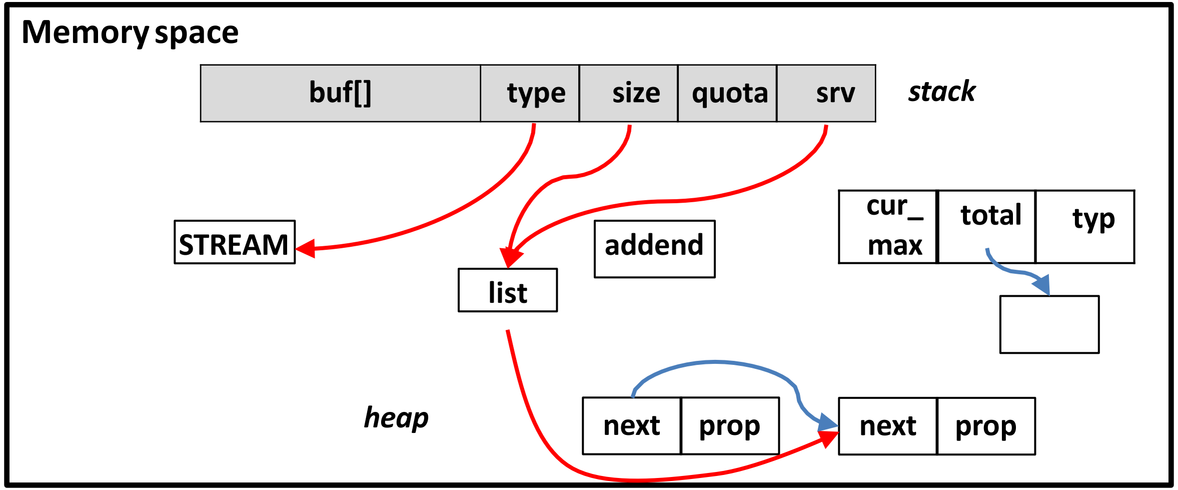
```
while (quota--) {
  readData(sockfd, buf);
  if(*type == NONE ) break;
  if(*type == STREAM)
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;
      srv->total += *size;
  }
}
```

vulnerable program

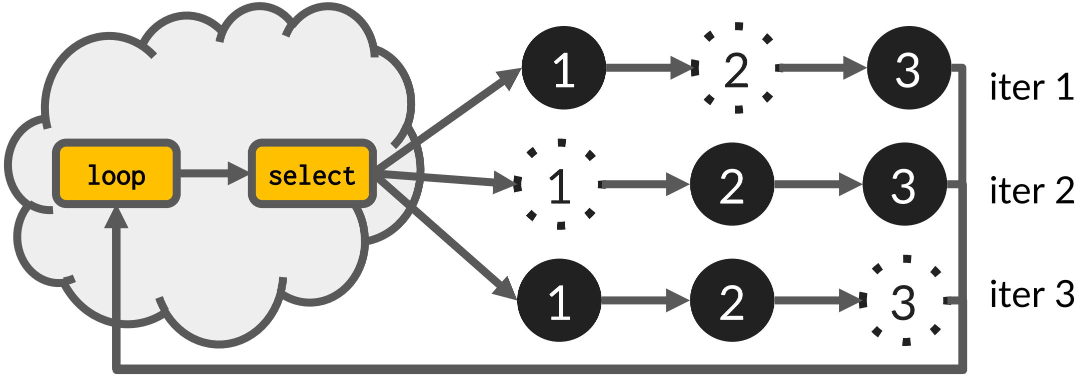
```
for(; list != NULL; list = list->next)
   list->prop += addend;
```

malicious computation

simulate ✓



DOP Gadget Dispatcher



Chain DOP gadgets **legitimately**

- `loop` - repeatedly invoke gadgets
- `select` - selectively activate gadgets

```
while (quota--) {                  // loop
  readData(sockfd, buf);           // selector
  if(*type == NONE ) break;
  if(*type == STREAM) *size = *(srv->cur_max);
  else { srv->typ = *type;srv->total += *size; }
}
```

MinDOP Language

Semantics	Statements in C	Data-Oriented Gadgets in DOP
arithmetic / logical	a op b	*p op *q
assignment	a = b	*p = *q
load	a = *b	*p = **q
store	*a = b	**p = *q
jump	goto L	vpc = &input
conditional jump	if (a) goto L	vpc &input if *p

p - &a; q - &b; op - any arithmetic / logical; vpc - virtual input pointer

DOP Demo

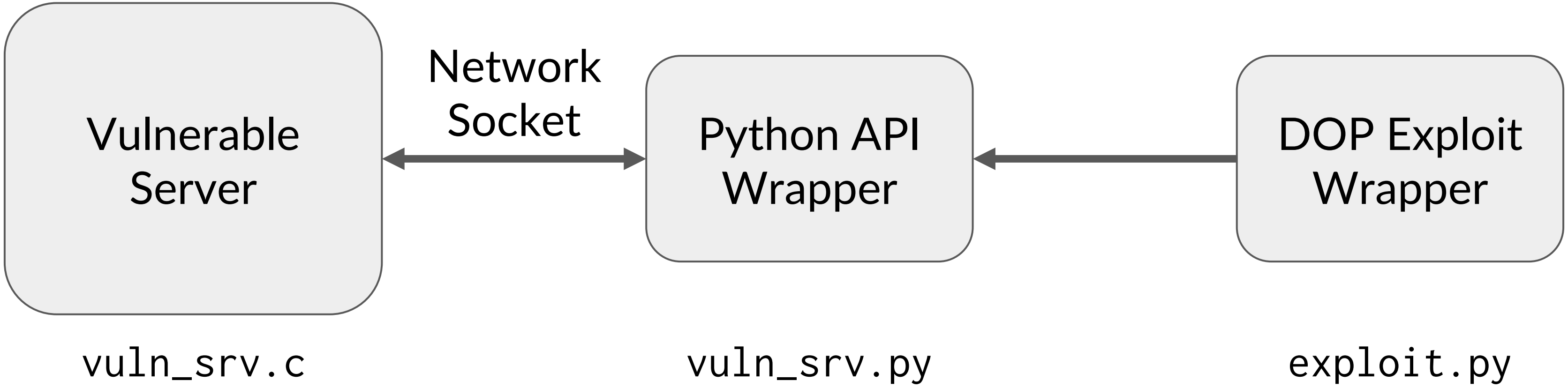
Minimal Vulnerability + Exploits

https://github.com/mayanez/min-dop

Extra: DOP Gadget Compiler
https://github.com/mayanez/llvm-clang-passes/tree/master/llvm/DOP-Gadgets

DOP Demo

Minimal Vulnerability + Exploits



General Architecture

DOP Demo

Leaking the SECRET

Steps

1. Find address holding SECRET.
2. Use DOP Load to fetch SECRET.
3. Exfiltrate SECRET.

DOP Demo

Leaking the SECRET

```
void do_serve(int sockfd) {...
// Memory Write Safety Violation
// Corrupts variables
// (ie. p_type, p_srv, etc)
readInData(g_clfd, sbuf);
...
else if (*p_type == TYPE_GET) {
      printf("[do_serve] TYPE_GET\n");
      getG_A(g_clfd);
}...
else if (*p_type == TYPE_LOAD) {
      printf("[do_serve] TYPE_LOAD\n");
       // DOP: load
      *p_g_d = **(p_srv->pp_b);
}...}
```

vuln_srv.c

DOP Demo

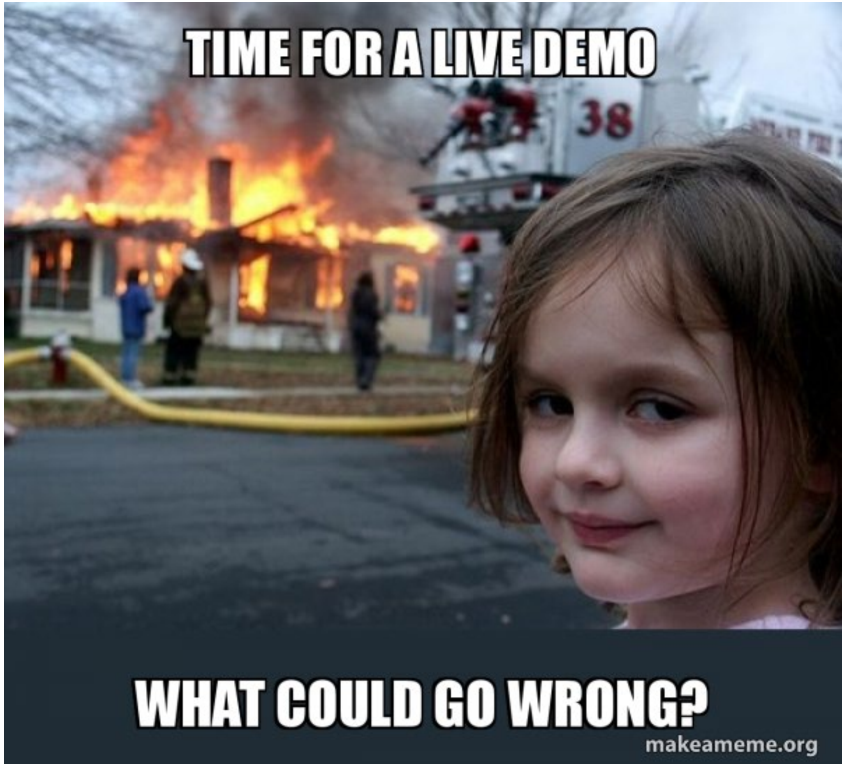
Leaking the SECRET

```
def dop_exfiltrate(self):
          ...
    # Equivalent: g_a = **g_pp_secret
    self.gadget_load(b, self._g_pp_secret__offset_base,
                     self._g_a__offset_base)

    # Equivalent: return g_a
    secret = self.vuln_srv.send_get()
    if secret == ExploitLib.SECRET: # SECRET = 0x1337
        return True
    else:
        return False
```

exploit.py

DOP Demo

Leaking the SECRET



Keep on Learning (More Data Attacks)

- Block-Oriented Programming (BOP)
  - An evolution of the original DOP technique.
  - [Arxiv:1805.04767] Block Oriented Programming: Automating Data-Only Attacks
- Survey on general Data-only attacks
  - [Arxiv:1902.08359] Exploitation Techniques and Defenses for Data-Oriented Attacks
    - Also includes discussion on defenses!

Memory Safety Going Forward

Memory Safety Going Forward

Defenses

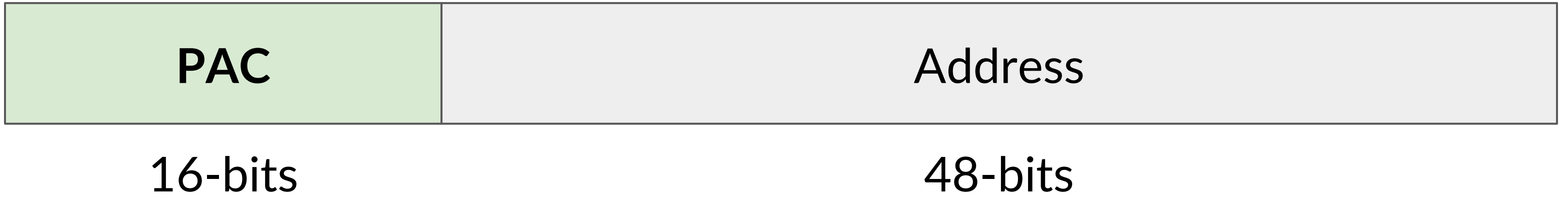
- **Hardware**
    - ARMv8.3 Pointer Authentication (PAC)

ARM

Memory Safety Going Forward

Defenses

- **Hardware**
  - ARMv8.3 Pointer Authentication (PAC)



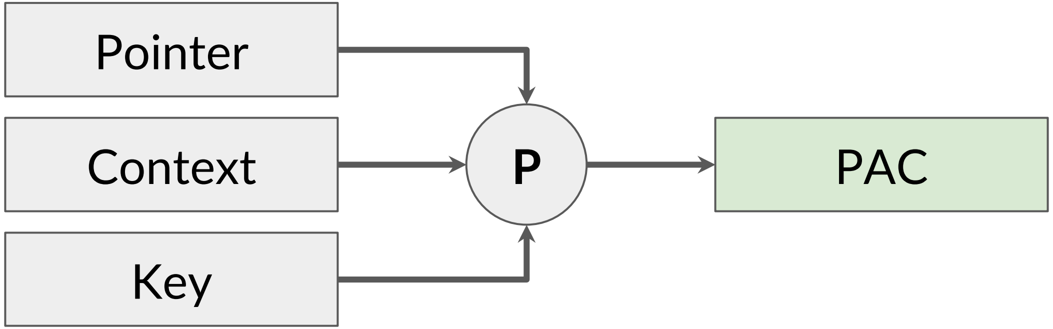
PAC	Address
16-bits	48-bits

- Pointer tagging via bits normally unused for virtual addressing.

Memory Safety Going Forward

Defenses

- **Hardware**
  - ARMv8.3 Pointer Authentication (PAC)



- PAC algorithm **P** is currently QARMA.

Memory Safety Going Forward

Defenses

- **Hardware**
  - ARMv8.3 Pointer Authentication (PAC)
  - Cryptographic CFI (CCFI)

Memory Safety Going Forward

Defenses

- **Hardware**
  - ARMv8.3 Pointer Authentication (PAC)
  - Cryptographic CFI (CCFI)
- **Languages**
  - Rust
    - See Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs

Memory Safety Going Forward

Defenses

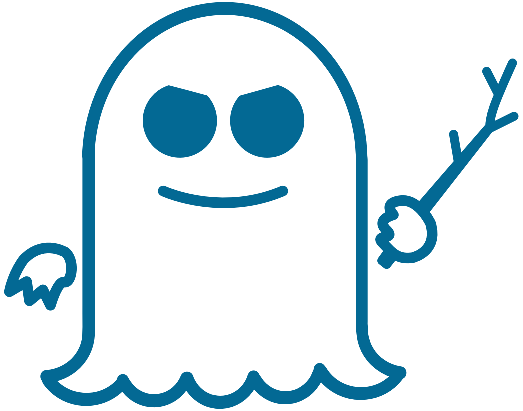
- **Hardware**
  - ARMv8.3 Pointer Authentication (PAC)
  - Cryptographic CFI (CCFI)
- **Languages**
  - Rust
    - See Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs
- **Compilers**
  - Sanitizers
    - See [Arxiv:1806.04355] SoK: Sanitizing for Security

Memory Safety Going Forward

Defenses

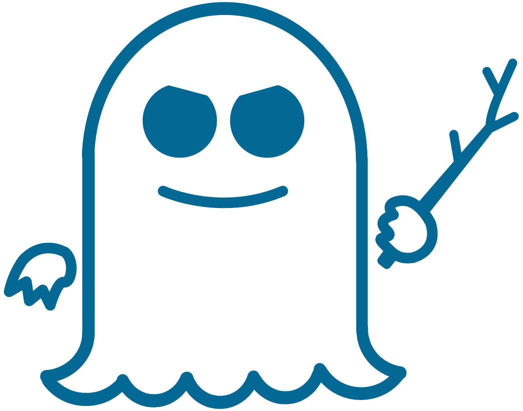
Memory Safety Going Forward

Defenses



Memory Safety Going Forward

Defenses



OFFENSIVECON BY Blue Frost Security

Check out: Popping Calc with Hardware Vulnerabilities

Memory Safety Going Forward

Defenses

Comprehensive

Composable

Questions?

Slides & Code can be found on my site:

https://miguel.arroyo.me/

@miguelaarroyo12