Heapster

Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images

***Fabio Gritti**, Fabio Pagani, Lukas Dresel, Ilya Grishchenko, Nilo Redini, Christopher Kruegel, and Giovanni Vigna*

University of California, Santa Barbara

Security of Dynamic Allocators for Monolithic Firmware Images

Security of Dynamic Allocators for Monolithic Firmware Images

Dynamic Allocators

- Dynamic allocators are algorithms used to manage dynamic memory (i.e., heap memory) of programs.

Dynamic Allocators

- Dynamic allocators are algorithms used to manage dynamic memory (i.e., heap memory) of programs.

- A robust dynamic allocator algorithm is crucial for the performance of any software.

Dynamic Allocators

- Dynamic allocators are algorithms used to manage dynamic memory (i.e., heap memory) of programs.
- A robust dynamic allocator algorithm is crucial for the performance of any software.
- Composed by at least 2 primitives: malloc and free (i.e., Heap Management Library)

Dynamic Allocators

- Dynamic allocators are routinely abused as building blocks for complex exploits

(Pwn2Own) Zoom Heap based Buffer Overflow Remote Code Execution Vulnerability

ZDI-21-971
ZDI-CAN-13587

Dynamic Allocators

- Dynamic allocators are routinely abused as building blocks for complex exploits

(Pwn2Own) Zoom Heap based Buffer Overflow Remote Code Execution Vulnerability

ZDI-21-971
ZDI-CAN-13587

Zero Day Initiative
@thezdi

Our 1st #Pwn2Own #AfterDark concludes with the Mofoffensive Research Team combining a heap overflow and a stack-based buffer overflow to gain code execution on the LAN interface of the NETGEAR R6700 router. Their efforts earn $5,000 and 1 Master of Pwn point. #P2OAustin

Dynamic Allocators

- Dynamic allocators are routinely abused as building blocks for complex exploits

(Pwn2Own) Zoom Heap based Buffer Overflow Remote Code Execution Vulnerability

ZDI-21-971
ZDI-CAN-13587

Zero Day Initiative
@thezdi

Our 1st #Pwn2Own #AfterDark concludes with the Mofoffensive Research Team combining a heap overflow and a stack-based buffer overflow to gain code execution on the LAN interface of the NETGEAR R6700 router. Their efforts earn $5,000 and 1 Master of Pwn point. #P2OAustin

SECURITY RESEARCH

CVE-2021-43267: Remote Linux Kernel Heap Overflow | TIPC Module Allows Arbitrary Code Execution

MAX VAN AMERONGEN / NOVEMBER 4, 2021

Dynamic Allocators

- Dynamic allocators are routinely abused as building blocks for complex exploits

Current research is focused on allocators for "classic" systems

Zero Day Initiative
@thezdi

Our 1st #Pwn2Own #AfterDark concludes with the Mofoffensive Research Team combining a heap overflow and a stack-based buffer overflow to gain code execution on the LAN interface of the NETGEAR R6700 router. Their efforts earn $5,000 and 1 Master of Pwn point. #P2OAustin

SECURITY RESEARCH

CVE-2021-43267: Remote Linux Kernel Heap Overflow | TIPC Module Allows Arbitrary Code Execution

MAX VAN AMERONGEN / NOVEMBER 4, 2021

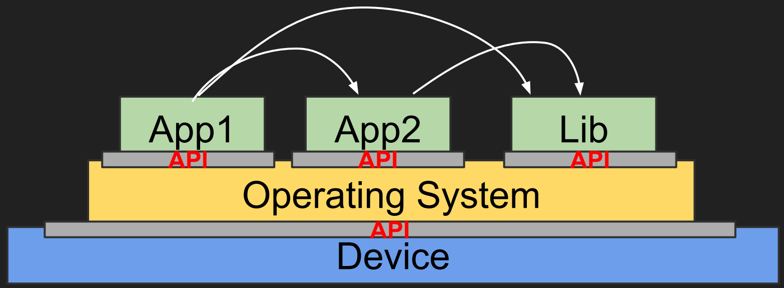
Security of Dynamic Allocators for Monolithic Firmware Images

Monolithic Firmware Images

- Firmware images without an OS-abstraction

Monolithic Firmware Images

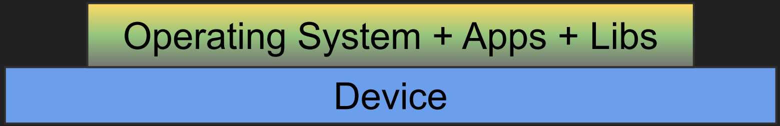
- Firmware images without an OS-abstraction



[Linux-based firmware image]

Monolithic Firmware Images

- Firmware images without an OS-abstraction



[Monolithic firmware image]

Monolithic Firmware Images

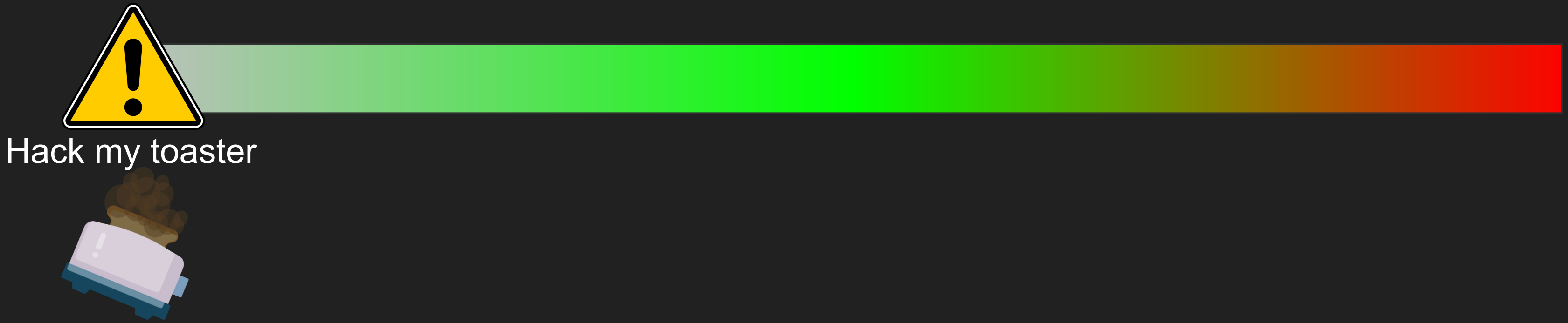
Empower a huge amount of diverse IoT devices

Monolithic Firmware Images

Empower a huge amount of diverse IoT devices

HUGE attack surface with a spectrum of different threat scenarios

Monolithic Firmware Images



Monolithic Firmware Images

Remote arbitrary write over pacemaker…

Monolithic Firmware Images

- Very hard target for both static and dynamic analysis:
  - Binary ONLY
  - NO symbols
  - NO hardware
  - Scalability of re-hosting remains a challenge

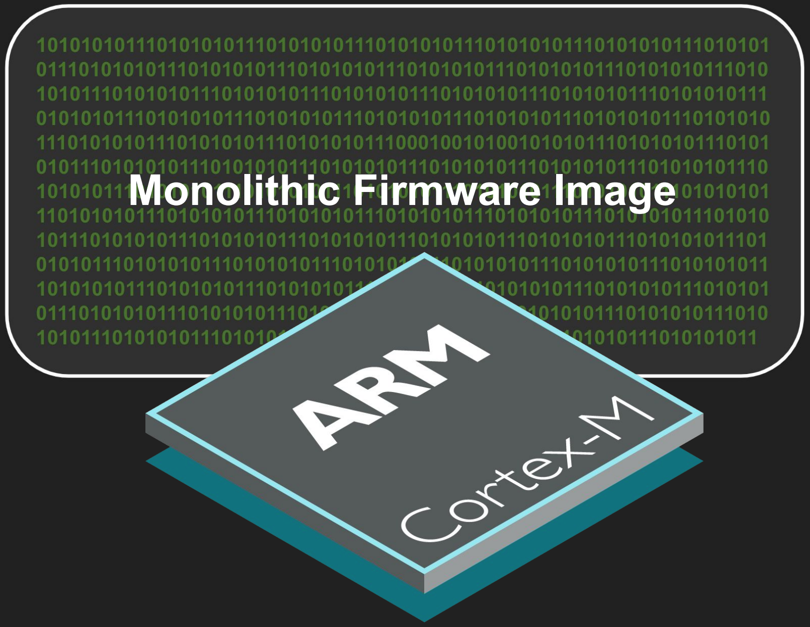

Monolithic firmware
Dynamic allocators

🤔 Understand if the f/w binary is using a dynamic allocator?

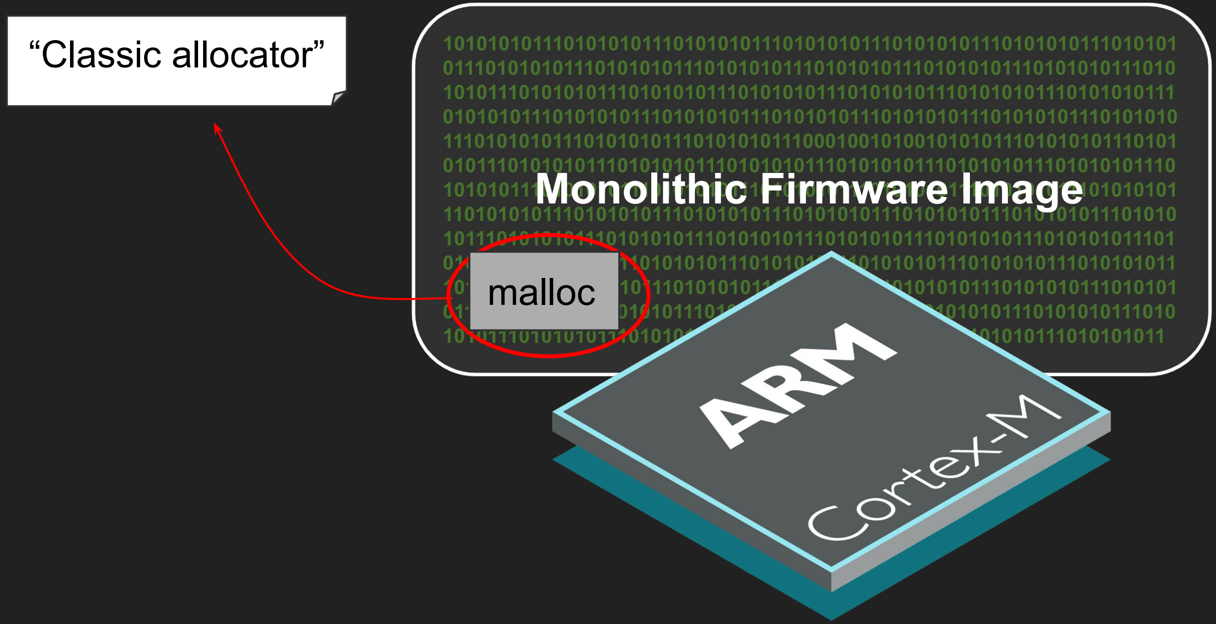
🤔 Understand if the f/w binary is using a dynamic allocator?

🤔 Check if the f/w allocator is robust against attacks?

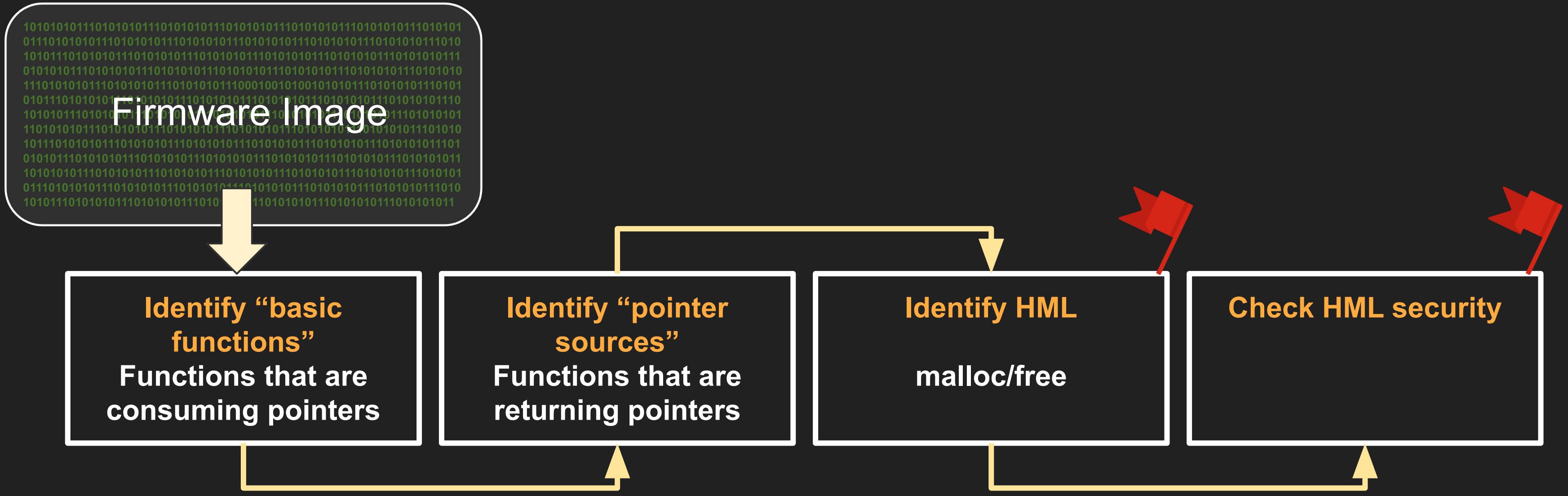
Research Scope



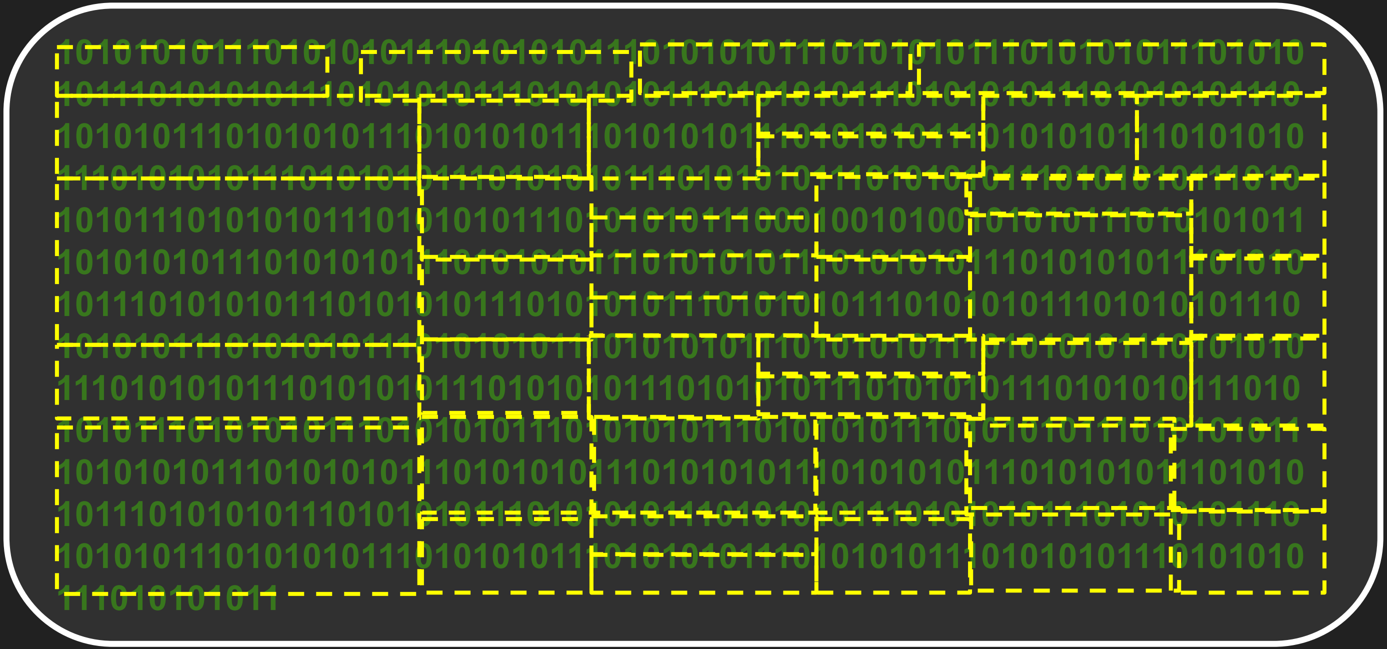
Research Scope

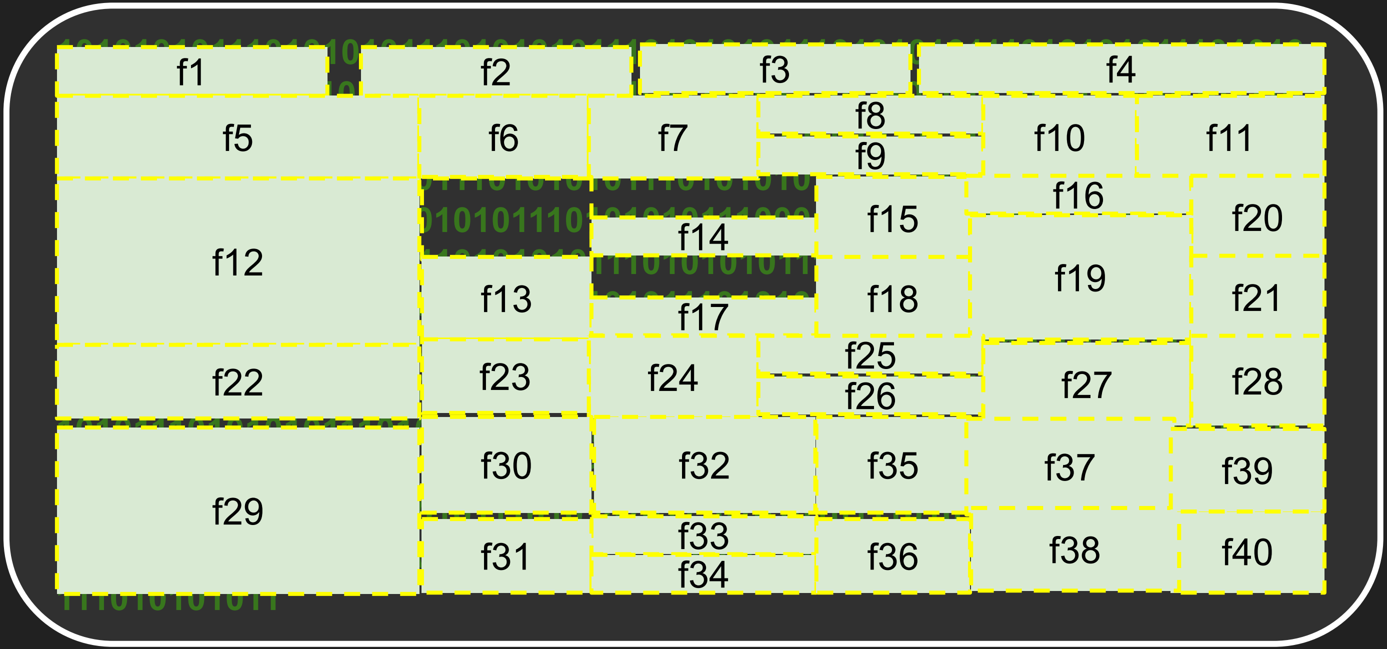


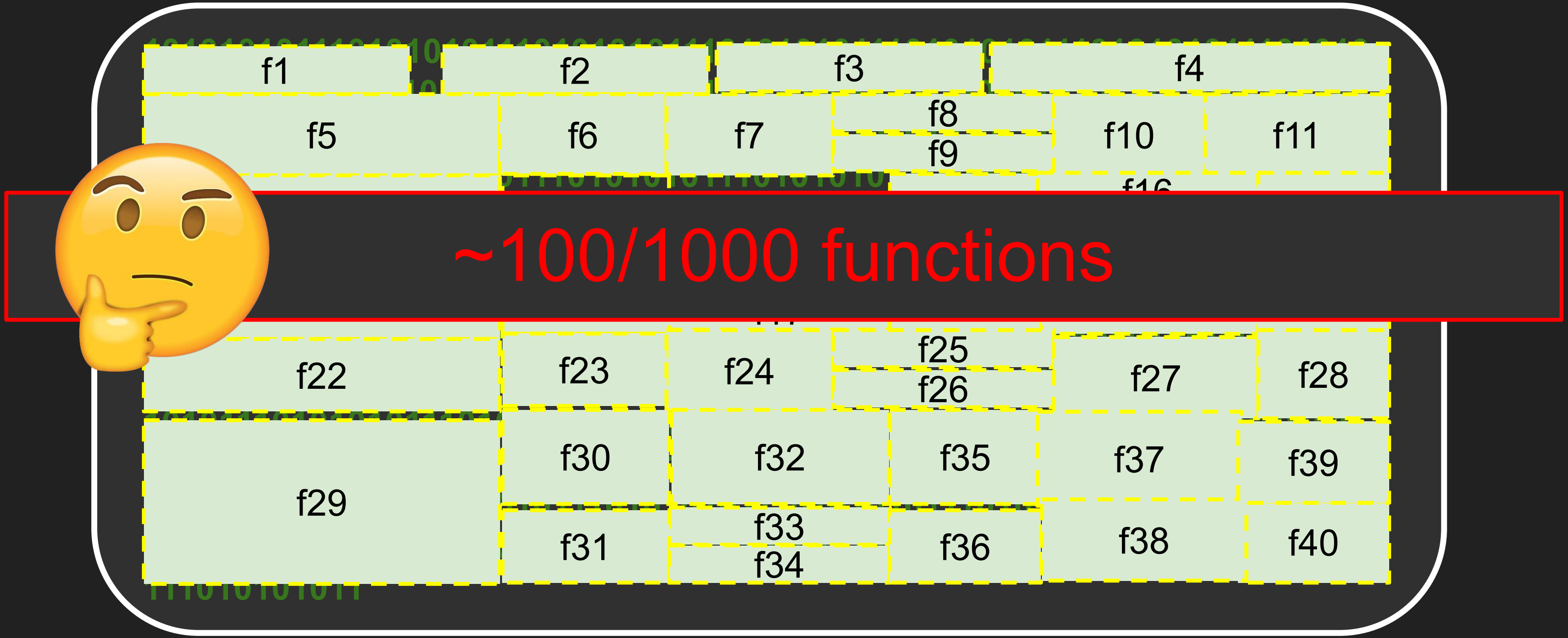
Heapster


Firmware Image
Identify "basic functions"
Functions that are consuming pointers
Identify "pointer sources"
Functions that are returning pointers
Identify HML
malloc/free
Check HML security

101010101110101010111010101011101010101110101010111010101011101010
101110101010111010101011101010101110101010111010101011101010101110
101010111010101011101010101110101010111010101011101010101110101010
111010101011101010101110101010111010101011101010101110101010111010
101011101010101110101010111010101011100010010100101010111010101011
101010101110101010111010101011101010101110101010111010101011101010
101110101010111010101011101010101110101010111010101011101010101110
101010111010101011101010101110101010111010101011101010101110101010
111010101011101010101110101010111010101011101010101110101010111010
101011101010101110101010111010101011101010101110101010111010101011
101010101110101010111010101011101010101110101010111010101011101010
101110101010111010101011101010 01010111010101011101010101110
1010101110101010111010101011 010101011101010101110101010
111010101011




f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
f11
f12
f13
f14
f15
f16
f17
f18
f19
f20
f21
f22
f23
f24
f25
f26
f27
f28
f29
f30
f31
f32
f33
f34
f35
f36
f37
f38
f39
f40


f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
f11
~100/1000 functions
f22
f23
f24
f25
f26
f27
f28
f29
f30
f32
f35
f37
f39
f31
f33
f34
f36
f38
f40

Approach

Memory allocators generate pointers

Approach

Memory allocators generate pointers

Pointers are eventually used to perform memory operations

Who uses memory pointers?

Identify Basic Functions

- These functions use pointers
  - memcpy(addr1,addr2,size)
  - memset(addr1,c,size)
  - memcmp(addr1,addr2,size)

- Simple to identify (i.e., “basic functions”)

Identify Basic Functions

Memcpy?

```
f8(X,Y,Z){

    [ … CODE … ]

}
```

```
X → b'wxyz\0wxyz'
Y → b'asdf\0asdf'
Z = 9
```

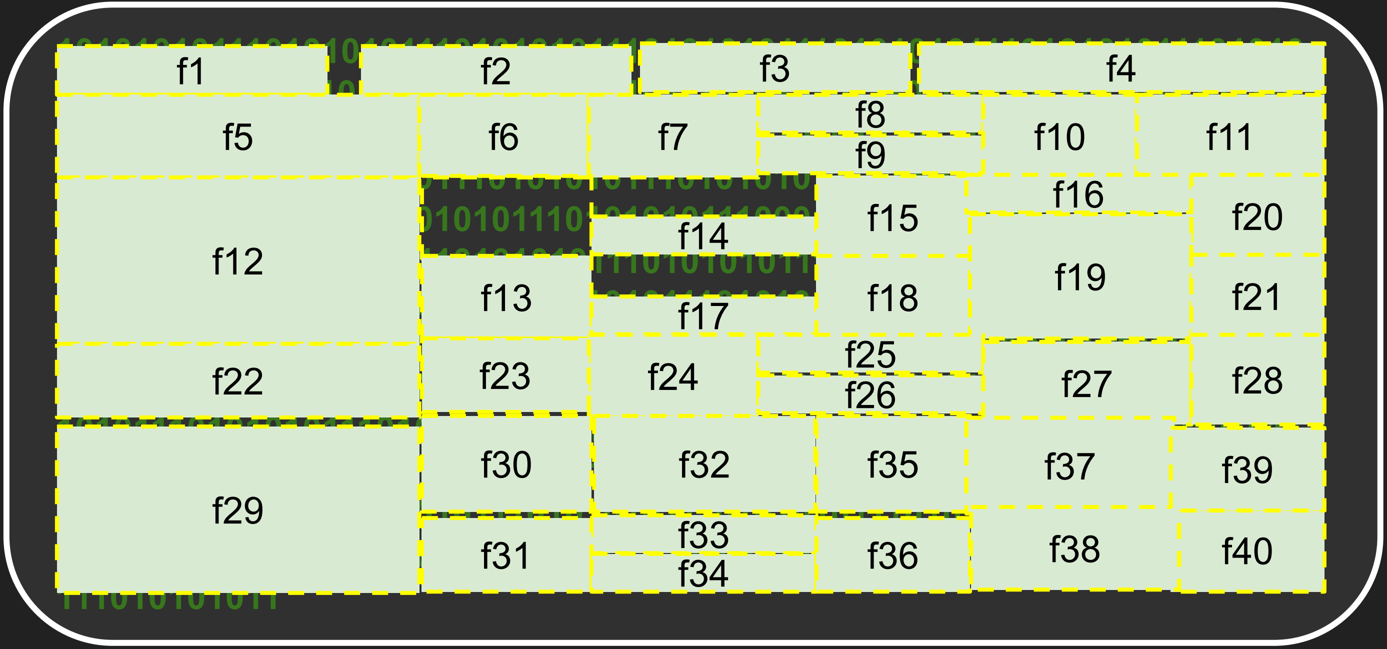
Identify Basic Functions

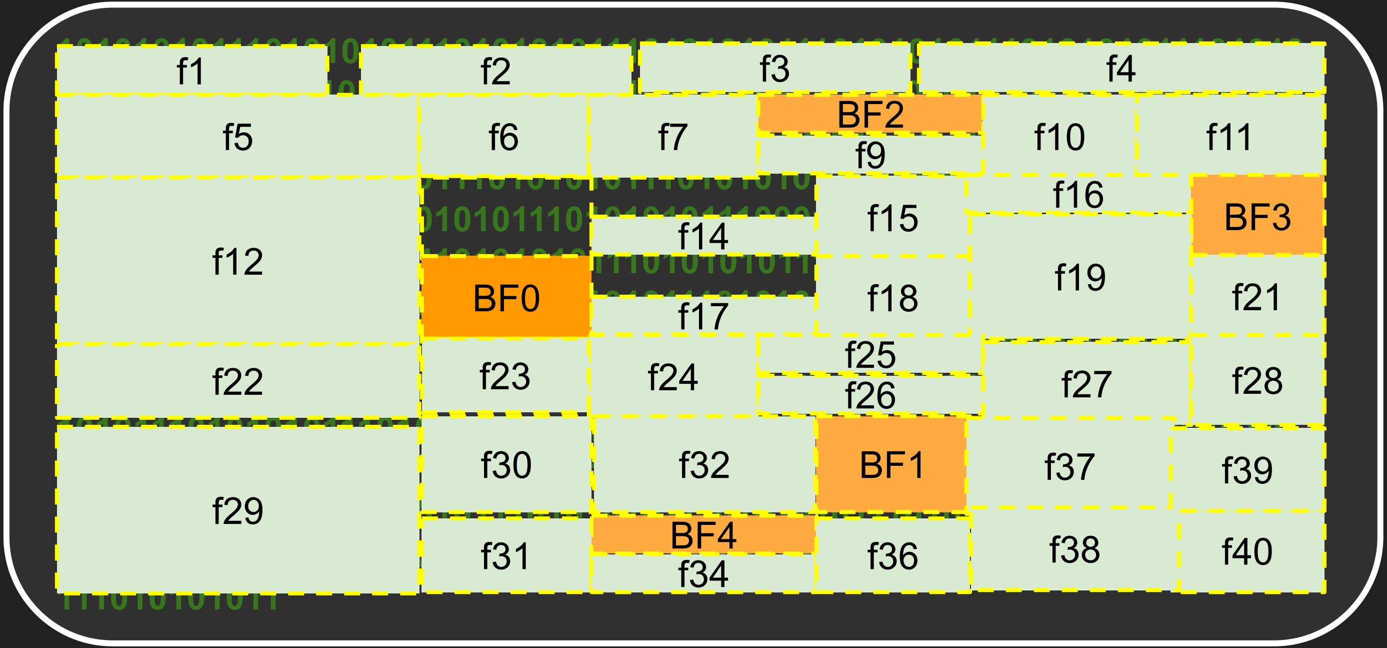
Memcpy?

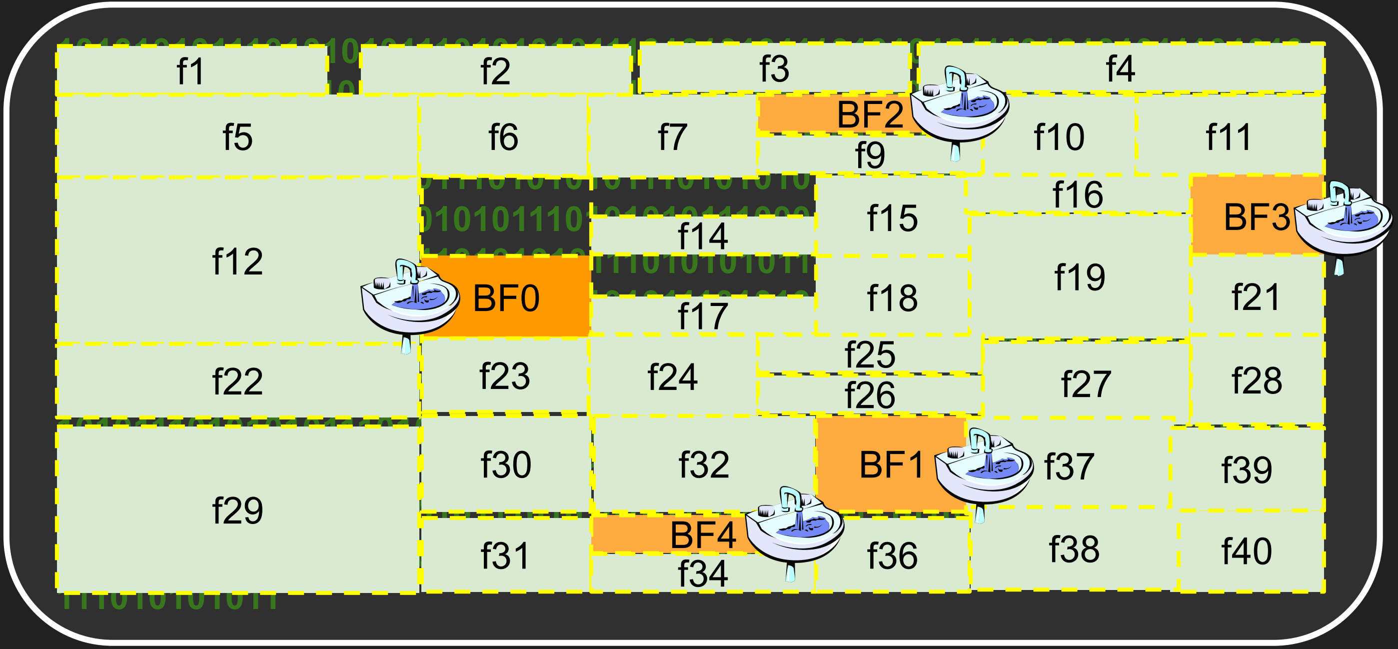
```
f8(X,Y,Z){

    [ ... CODE ... ]

}
```

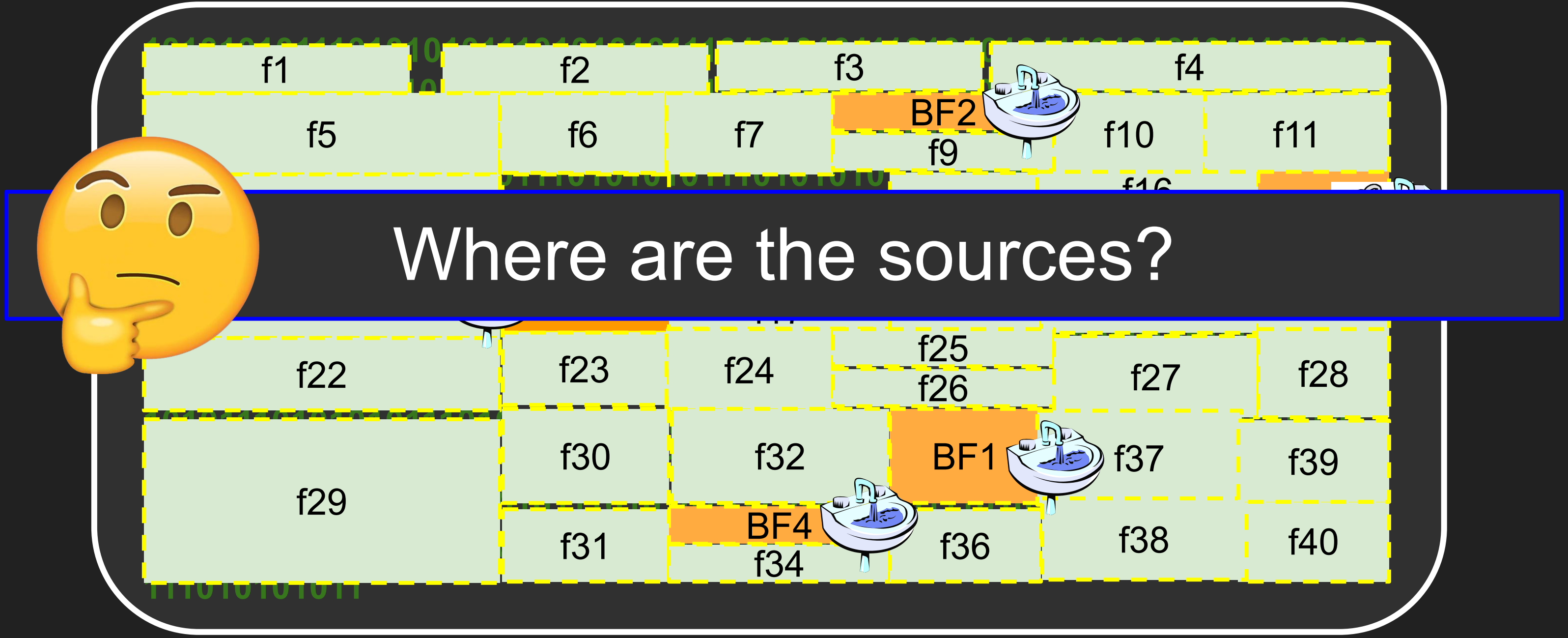
```
X → b'wxyz\0wxyz'
Y → b'asdf\0asdf'
Z = 9
```

→ Does buffer at X contain exactly 9 bytes `b'asdf\0asdf'`?
→ Is buffer at Y unchanged?


f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
f11
f12
f14
f15
f16
f20
f13
f17
f18
f19
f21
f22
f23
f24
f25
f26
f27
f28
f29
f30
f31
f32
f33
f34
f35
f36
f37
f38
f39
f40


f1
f2
f3
f4
f5
f6
f7
BF2
f9
f10
f11
f12
f14
f15
f16
BF3
BF0
f17
f18
f19
f21
f22
f23
f24
f25
f26
f27
f28
f29
f30
f32
BF1
f37
f39
f31
BF4
f34
f36
f38
f40

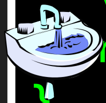

f1
f2
f3
f4
BF2
f5
f6
f7
f9
f10
f11
f12
f14
f15
f16
BF3
BF0
f17
f18
f19
f21
f22
f23
f24
f25
f26
f27
f28
f29
f30
f32
BF1
f37
f39
f31
BF4
f34
f36
f38
f40



Where are the sources?

Identify Pointer Sources

- Functions that provide arguments to the basic functions



```
v25 = (char *) f19(x);
v26 = v25;
if(v25){
   v25[28] = v27;
   v25[29] = 1;
   memcpy(v25 + 12, v21 + 2, 16)
}
```

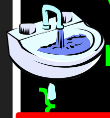
Identify Pointer Sources

```
v25 = (char *) f19(x);
v26 = v25;
if(v25){
   v25[28] = v27;
   v25[29] = 1;
   memcpy(v25 + 12, v21 + 2, 16)
}
```

- Use static taint engine (Reaching Definition)
- Collect *all* the functions that are returning values that define basic functions' arguments

Identify Pointer Sources

```
v25 = (char *) f19(x);
v26 = v25;
if(v25){
   v25[28] = v27;
   v25[29] = 1;
   memcpy(v25 + 12, v21 + 2, 16)
}
```

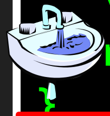


v25 defs:
{<f19_RETURN>}

- Use static taint engine (Reaching Definition)
- Collect *all* the functions that are returning values that define basic functions’ arguments

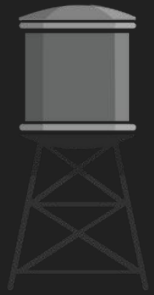
Identify Pointer Sources

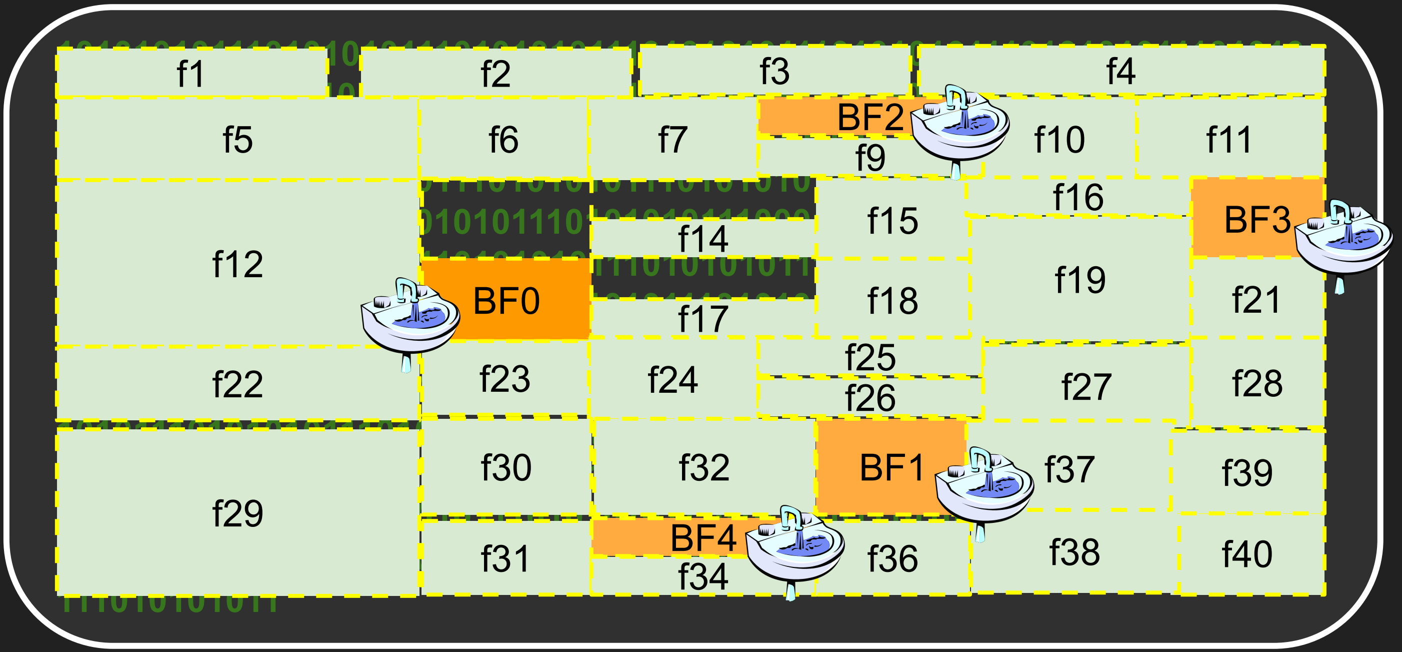
```
v25 = (char *) PS(x);
v26 = v25;
if(v25){
   v25[28] = v27;
   v25[29] = 1;
   memcpy(v25 + 12, v21 + 2, 16)
}
```



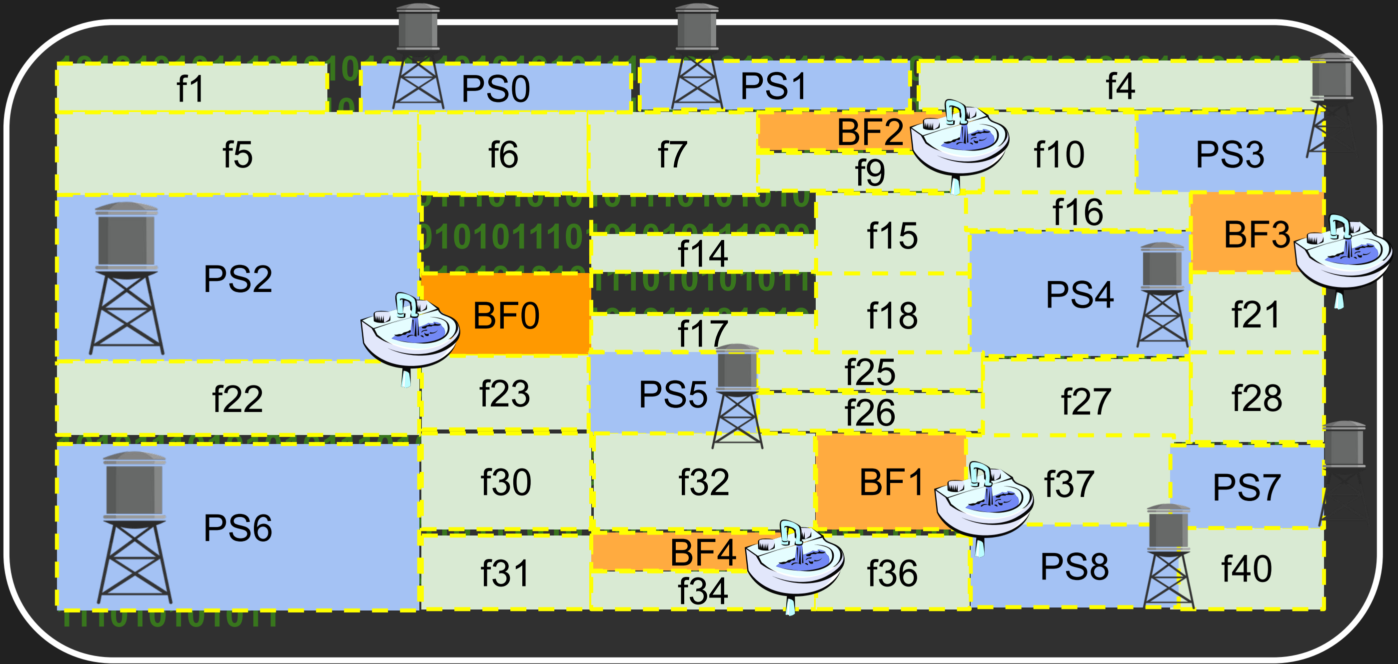
v25 defs:
{<f19_RETURN>}

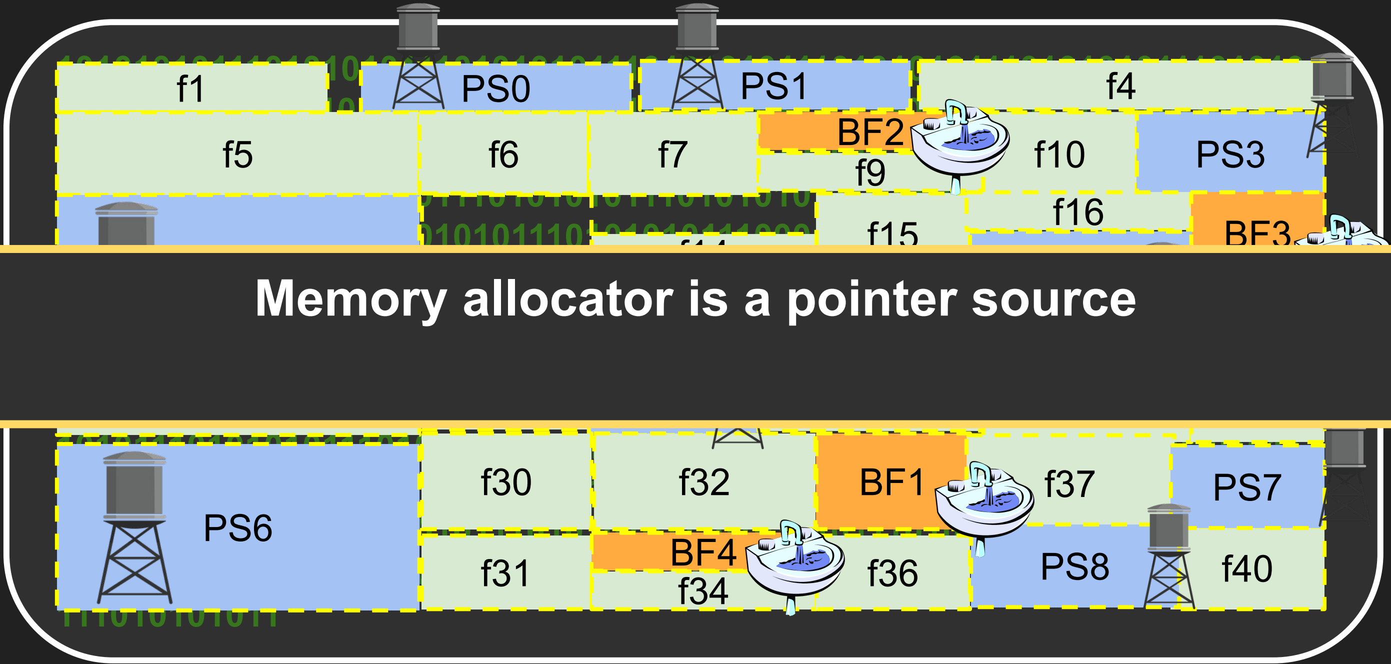
f19 is a Pointer Source!



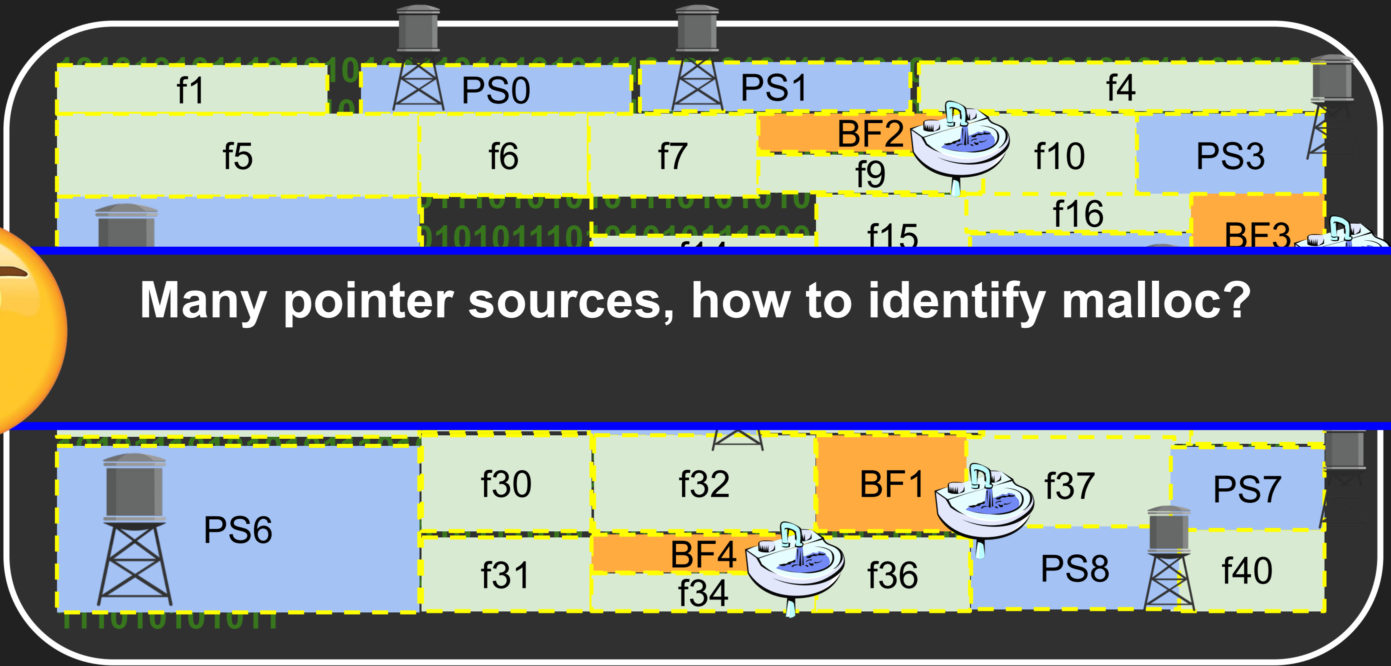


f1
f2
f3
f4
f5
f6
f7
BF2
f9
f10
f11
f12
BF0
f14
f15
f16
BF3
f17
f18
f19
f21
f22
f23
f24
f25
f26
f27
f28
f29
f30
f32
BF1
f37
f39
f31
BF4
f34
f36
f38
f40


f1
PS0
PS1
f4
f5
f6
f7
BF2
f9
f10
PS3
PS2
f14
f15
f16
BF3
BF0
f17
f18
PS4
f21
f22
f23
PS5
f25
f26
f27
f28
PS6
f30
f32
BF1
f37
PS7
f31
BF4
f34
f36
PS8
f40



Memory allocator is a pointer source



Many pointer sources, how to identify malloc?

Identify Malloc

Malloc returns pointers inside heap region

(CortexM: 0x20000000 -> 0x40000000)

Identify Malloc

Malloc returns pointers inside heap region

(CortexM: 0x20000000 -> 0x40000000)

Malloc returns different addresses to subsequent invocations

(Serve every request with a different memory block)

Identify Malloc

Identify Malloc

```
f(){
Call PS(0x5)
}

g(){
Call PS(0x12)
}

z(){
Call PS(0x7)
}
```

xref

xref

xref

```
PS(int x){

...

}
```

Identify Malloc

f(){
Call PS(0x5)
}

xref

PS(int x){

...

}

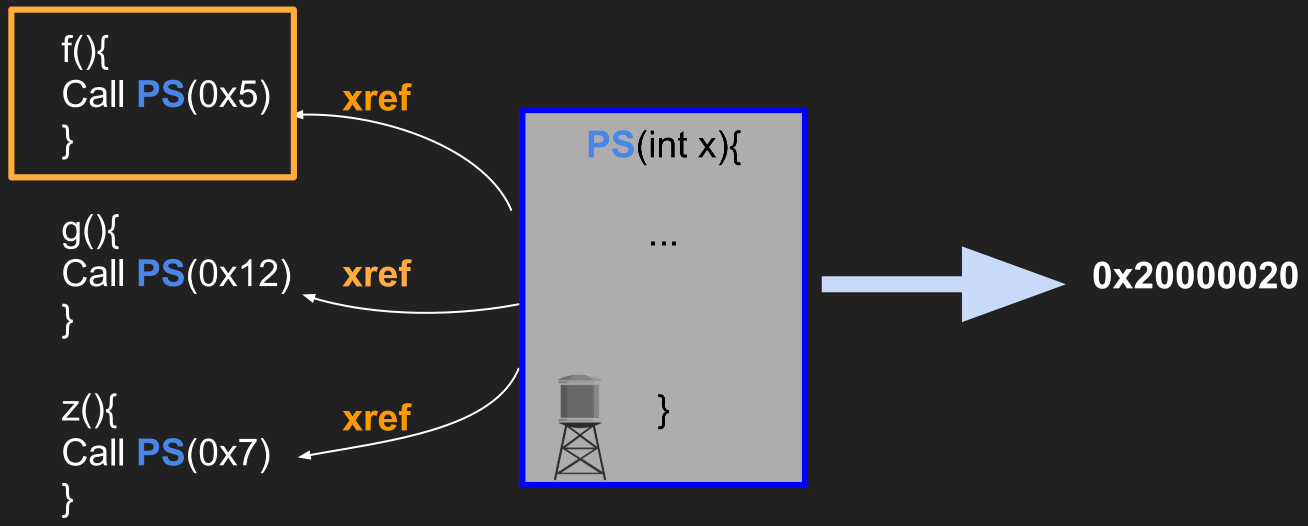
g(){
Call PS(0x12)
}

xref

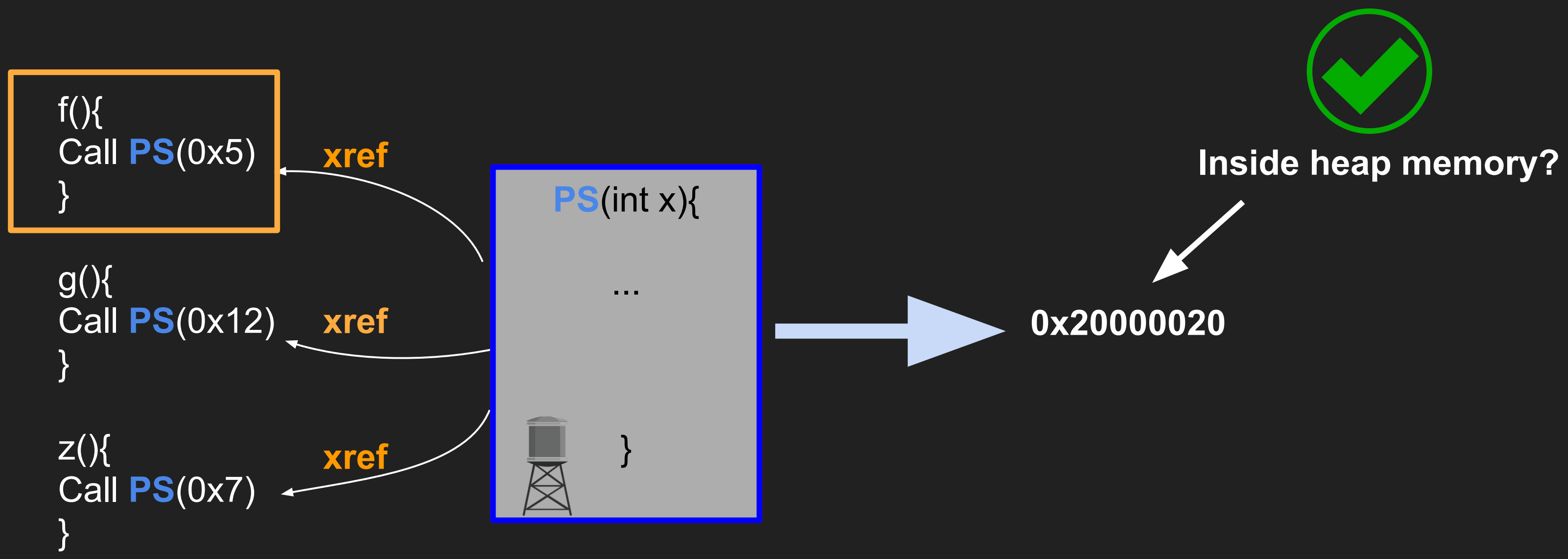
z(){
Call PS(0x7)
}

xref

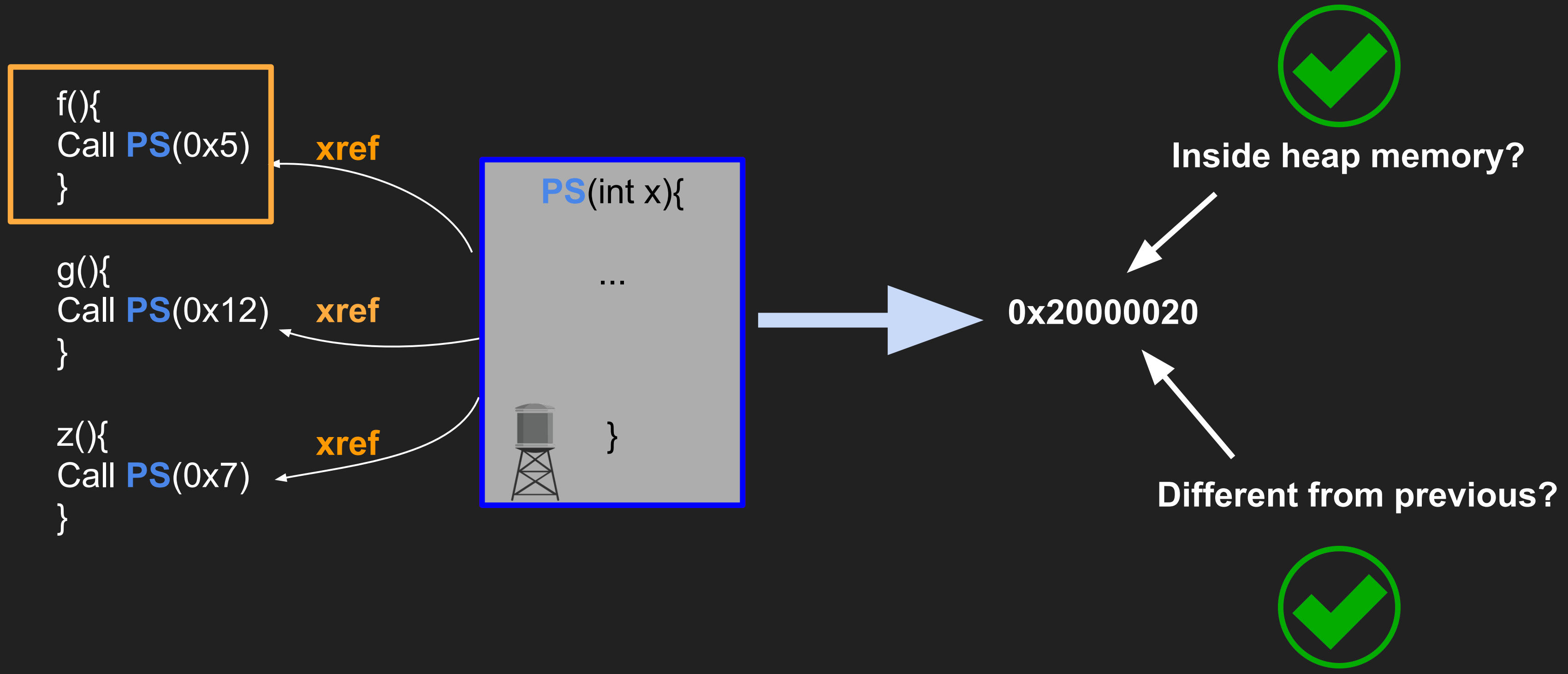
Identify Malloc



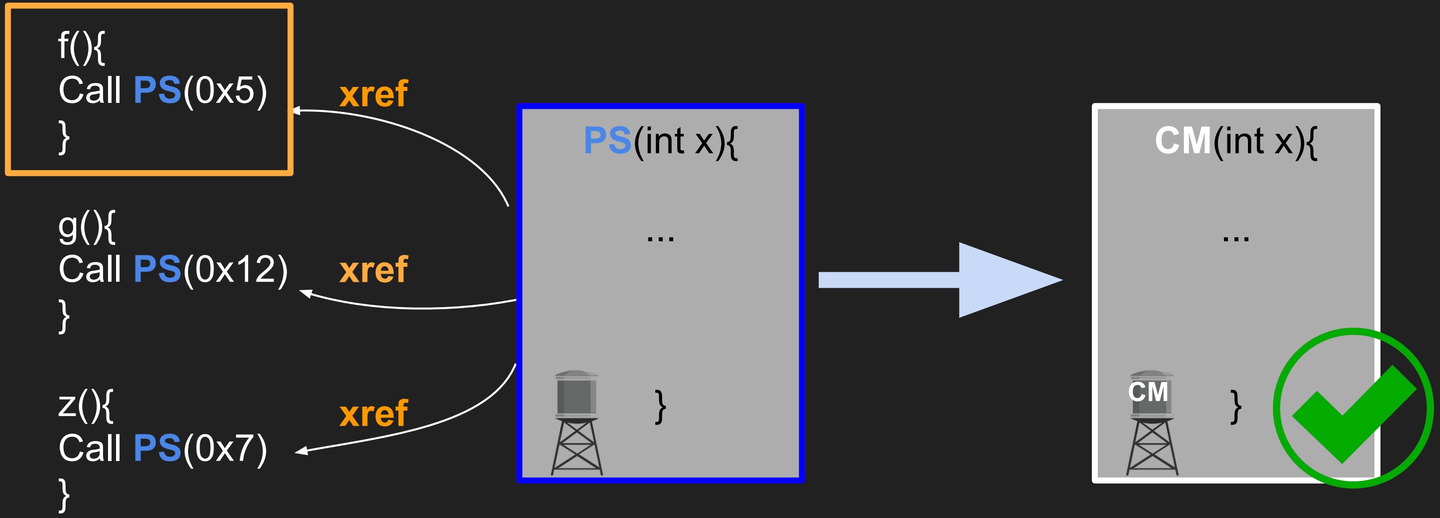
Identify Malloc



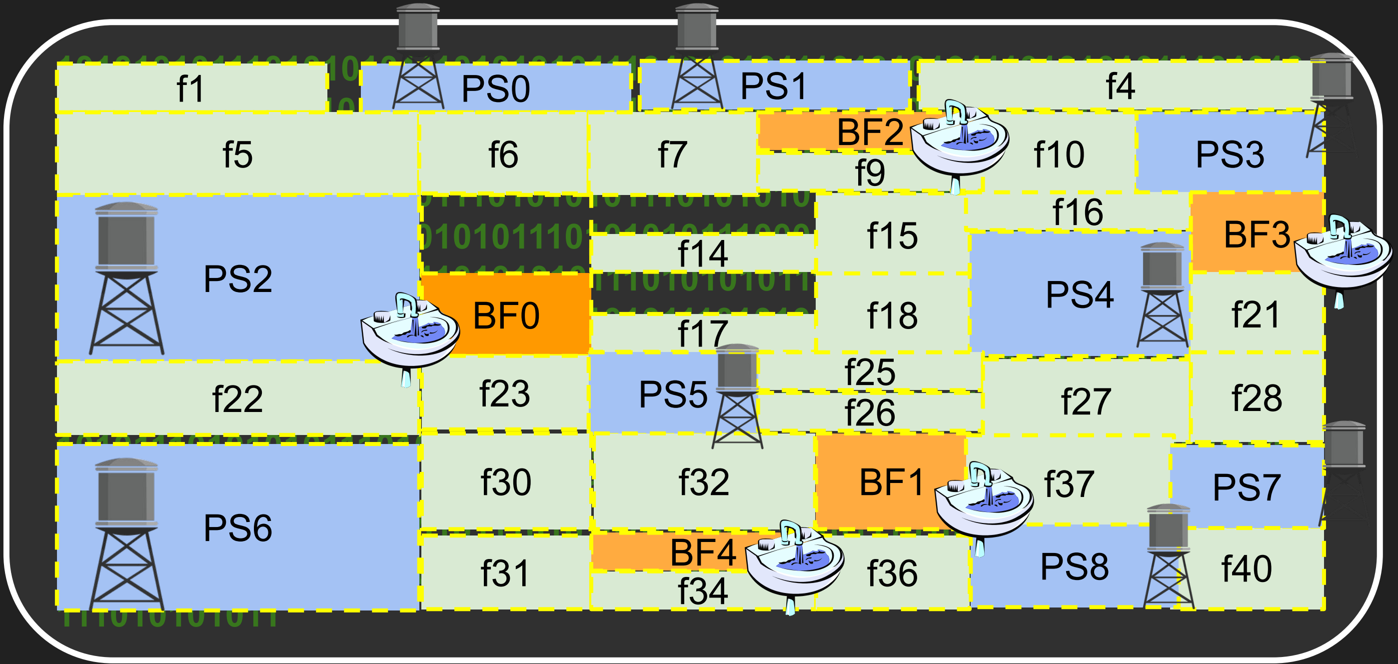
Identify Malloc

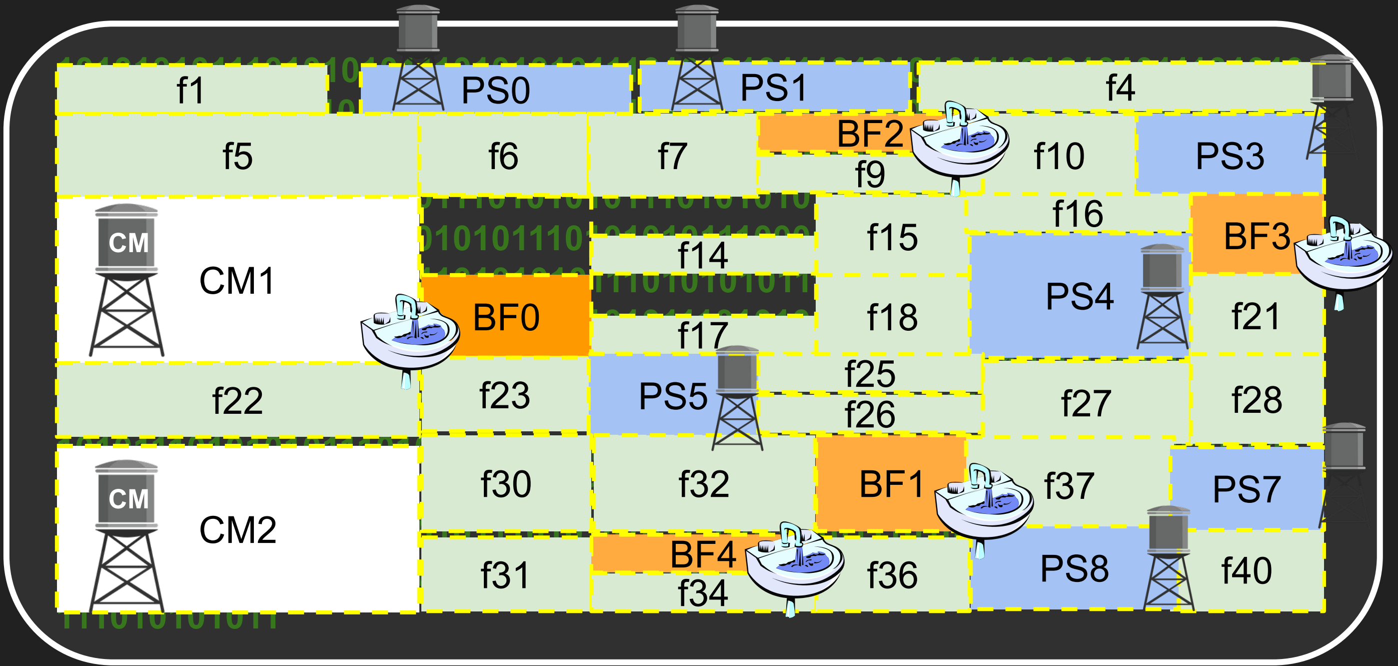


Identify Malloc

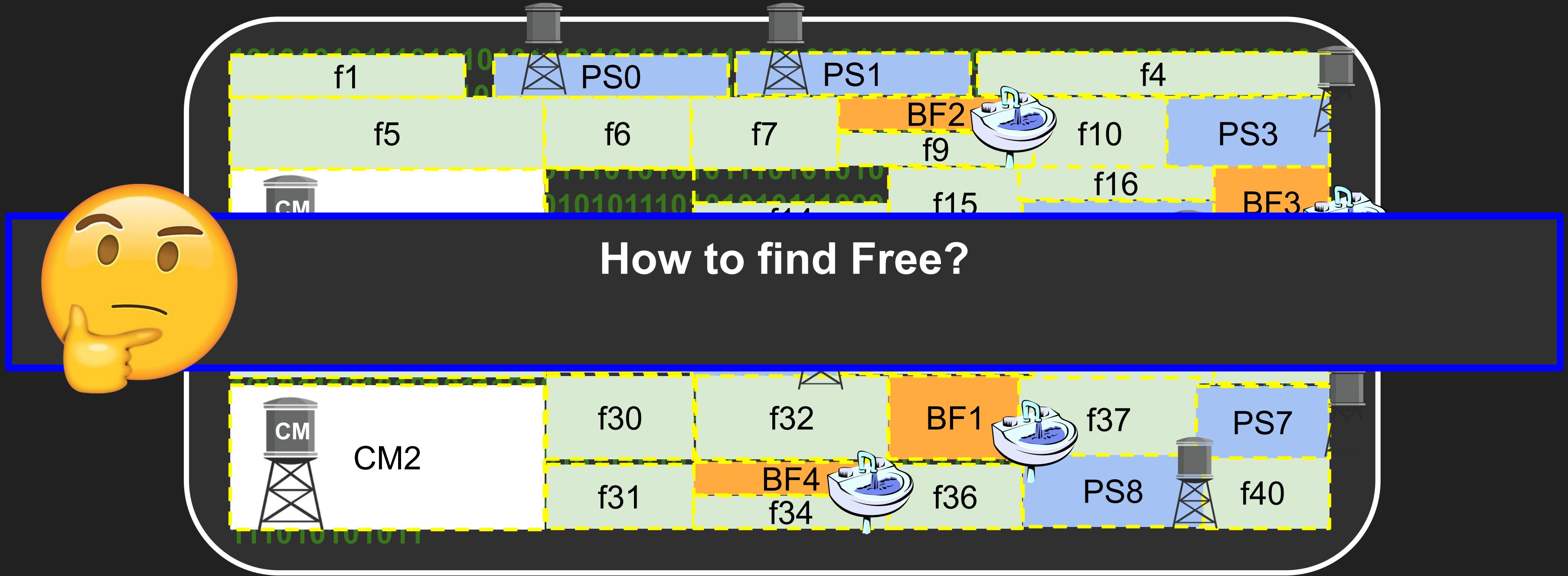


PS is a candidate malloc!


f1
PS0
PS1
f4
f5
f6
f7
BF2
f9
f10
PS3
PS2
f14
f15
f16
BF3
BF0
f17
f18
PS4
f21
f22
f23
PS5
f25
f26
f27
f28
PS6
f30
f32
BF1
f37
PS7
f31
BF4
f34
f36
PS8
f40



f1
PS0
PS1
f4
f5
f6
f7
BF2
f9
f10
PS3
CM
CM1
f14
f15
f16
BF3
BF0
f17
f18
PS4
f21
f22
f23
PS5
f25
f26
f27
f28
CM
CM2
f30
f32
BF1
f37
PS7
f31
BF4
f34
f36
PS8
f40

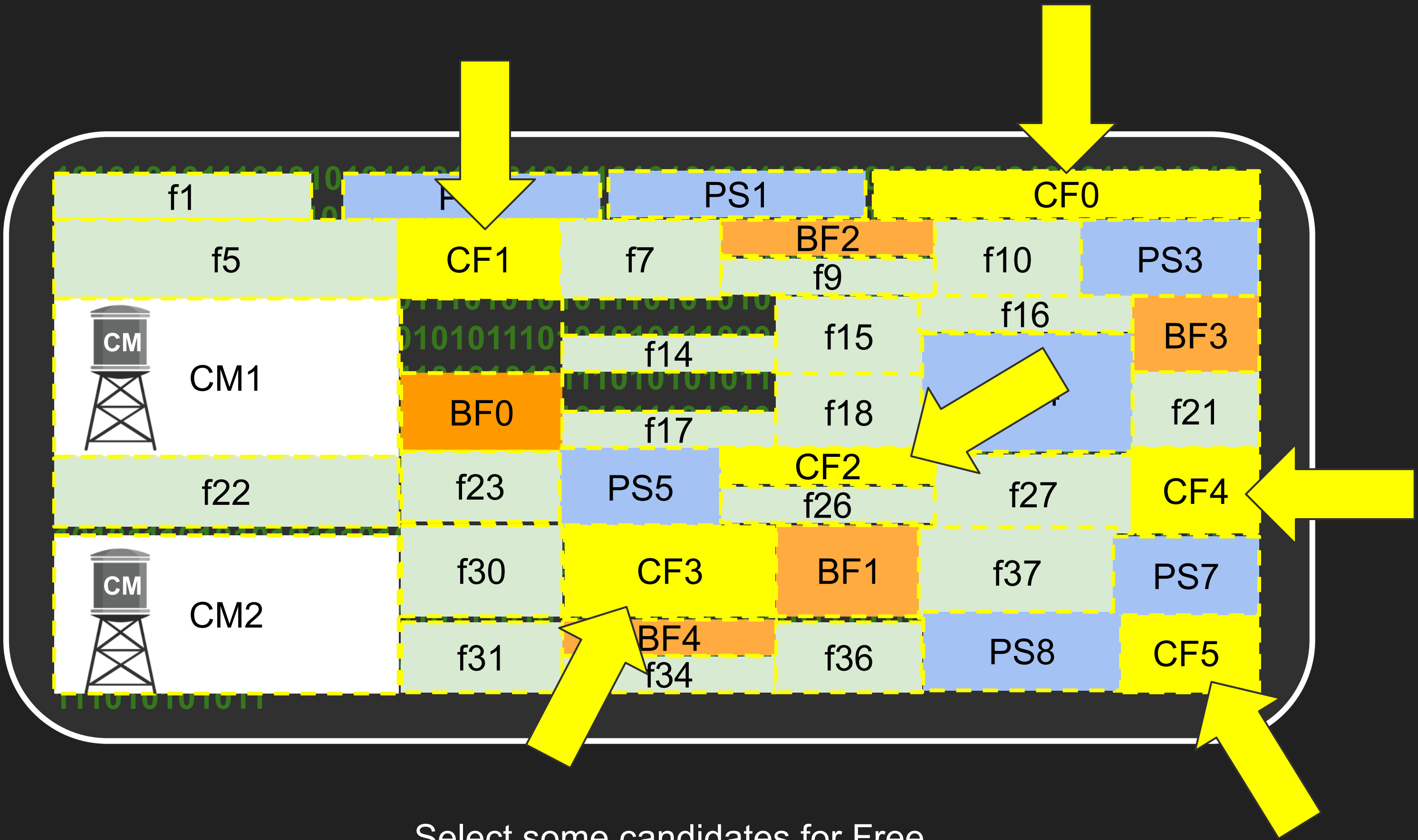


How to find Free?

Identify Free

Malloc and Free are LIFO

(malloc returns the last freed pointer)



Select some candidates for Free

Identify Free

CM(int x){

...

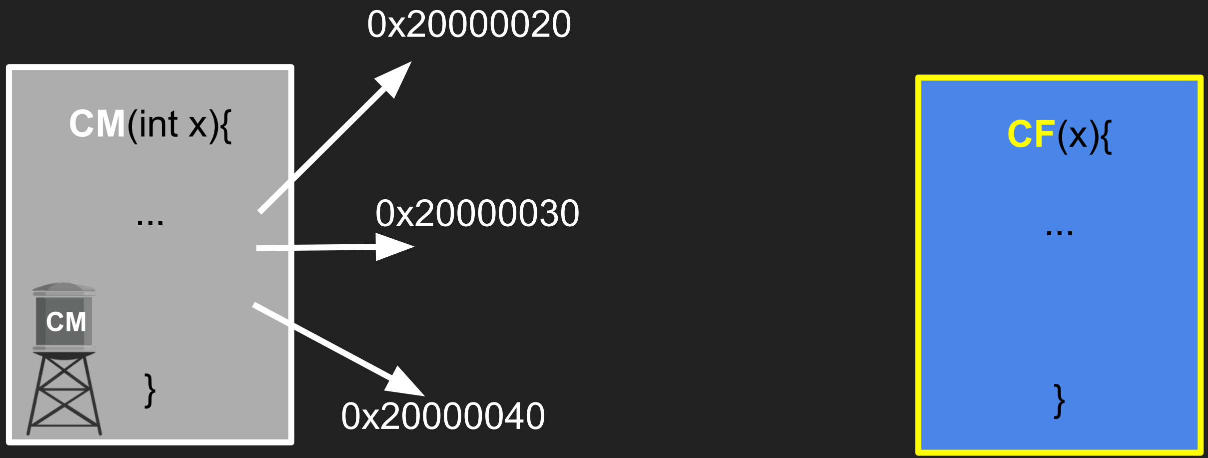
}

CF(x){

...

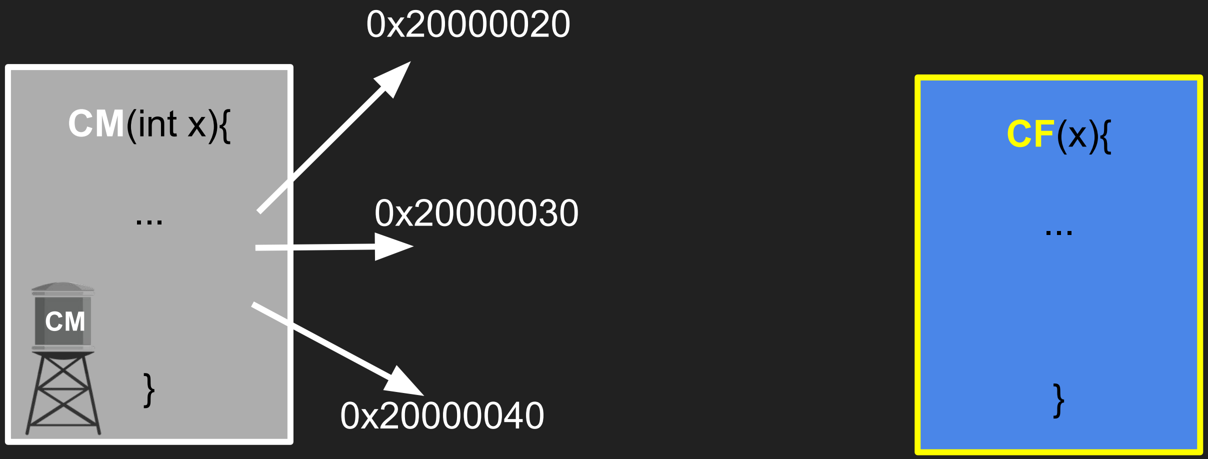
}

Identify Free



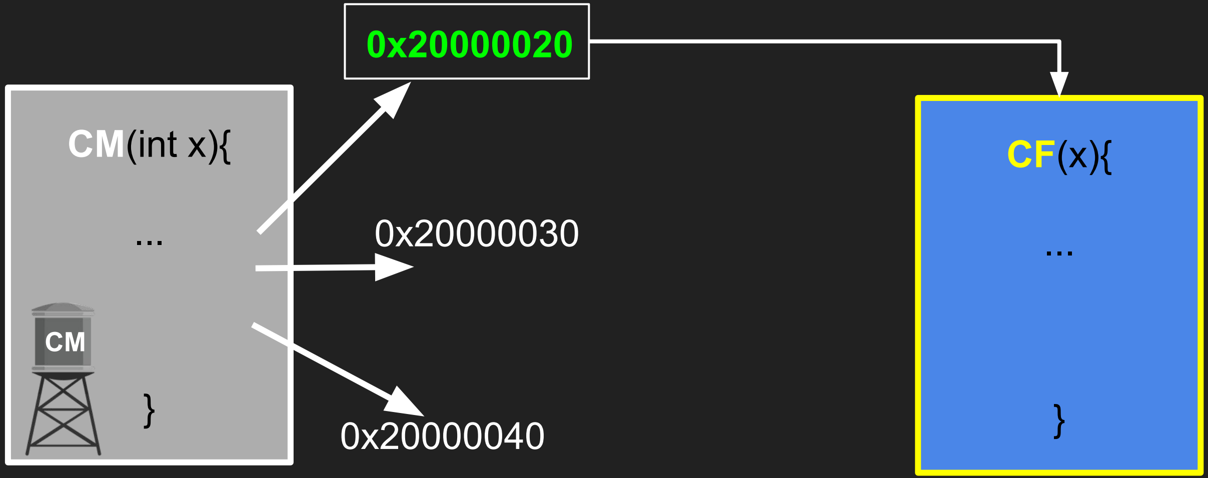
Call **CM** an X amount of times

Identify Free



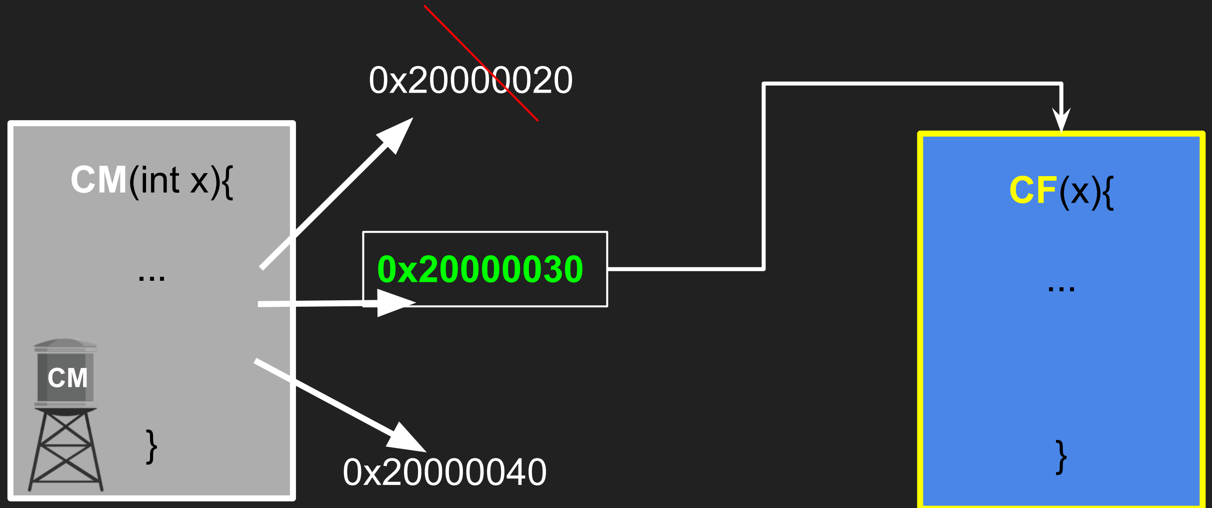
Allocated addresses= [0x20000020, 0x20000030, 0x20000040]

Identify Free



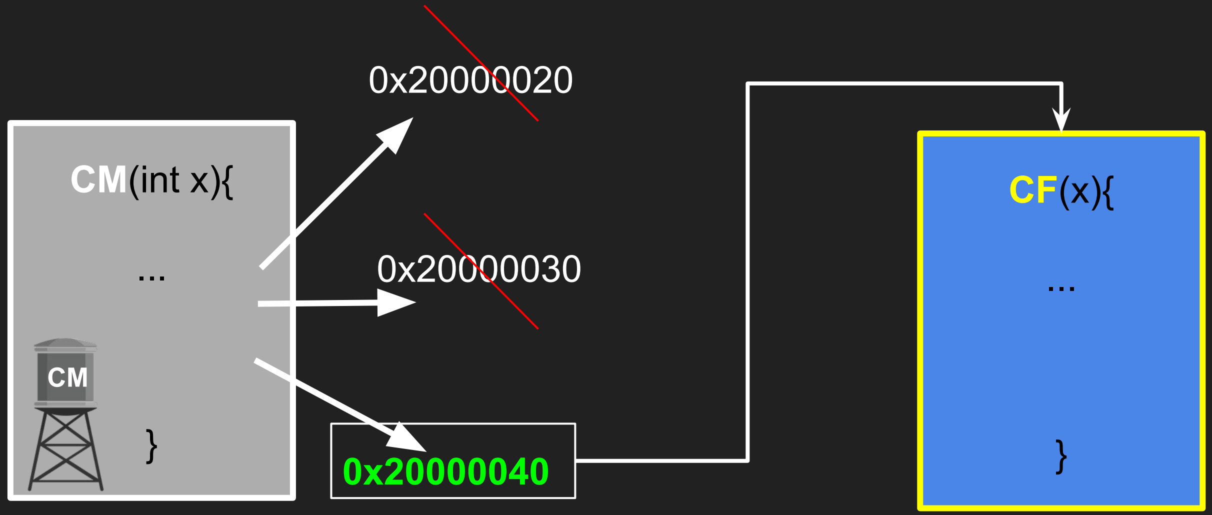
Allocated addresses= [0x20000020, 0x20000030, 0x20000040]

Identify Free



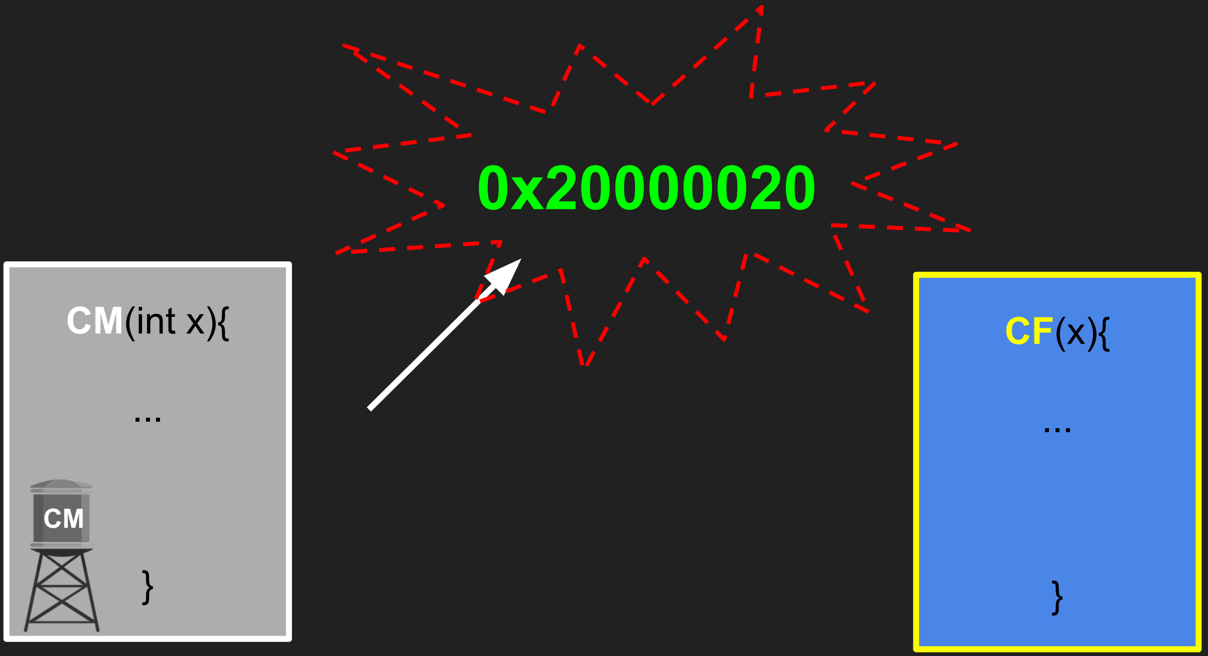
Allocated addresses= [0x20000020, 0x20000030, 0x20000040]

Identify Free



Allocated addresses= [0x20000020, 0x20000030, 0x20000040]

Identify Free



Call **CM** again

Allocated addresses= [**0x20000020**, 0x20000030, 0x20000040]

Identify Free

```
CM(int x){

    ...

}
```

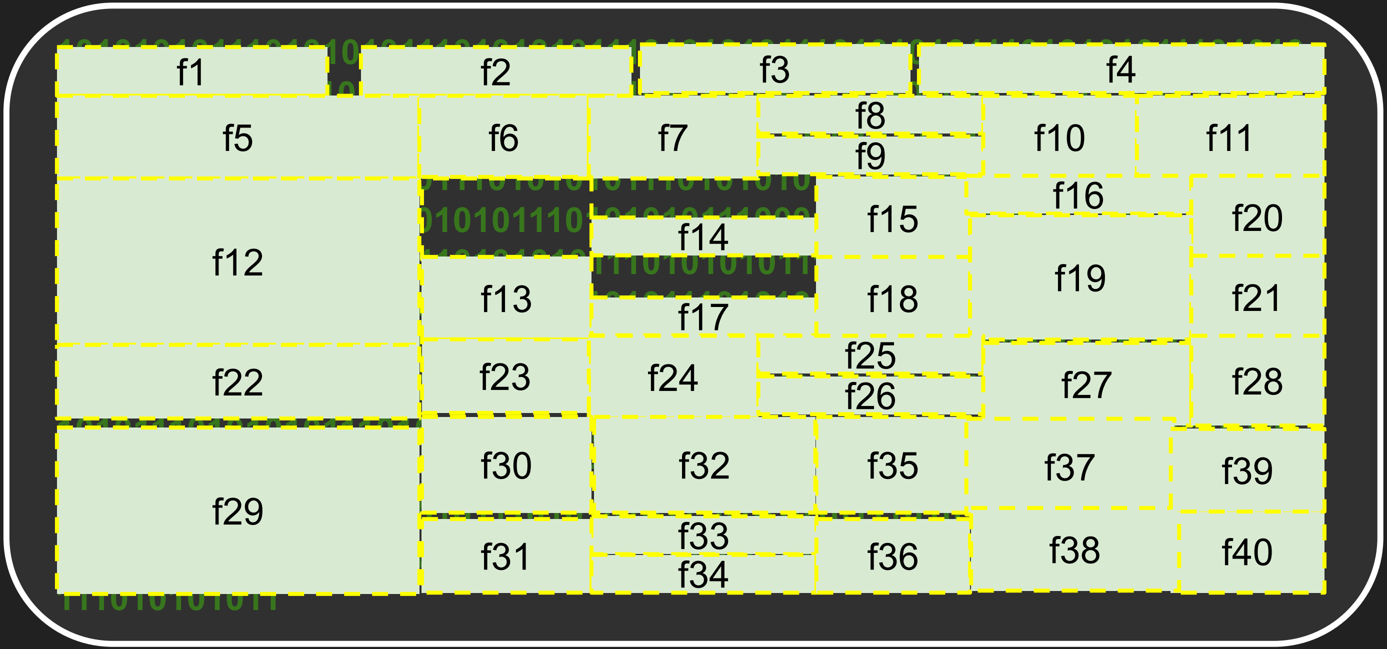

```
CF(x){

    ...

}
```

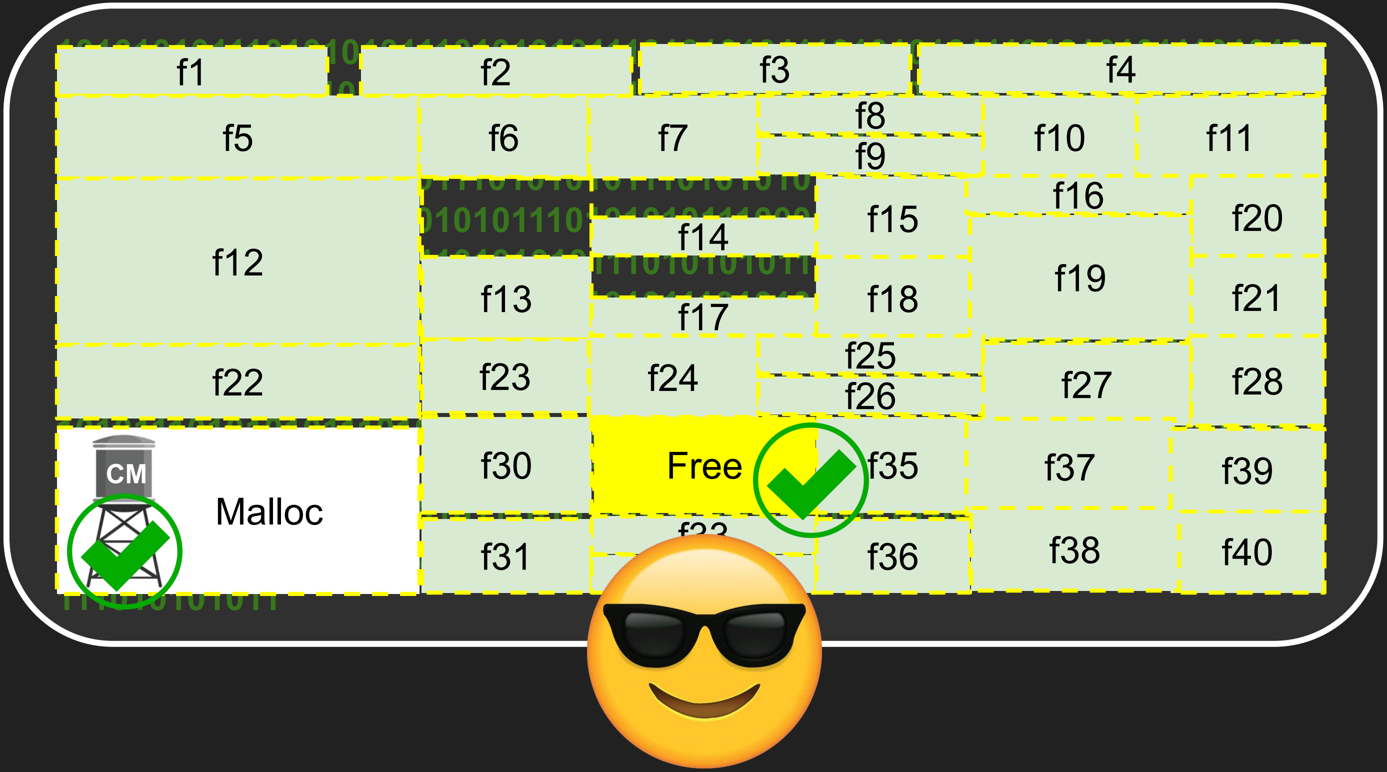
Identify Free

Heap Management Library (i.e., HML)

```
Malloc(int x){
    ...
}
```


```
Free(x){
    ...
}
```


f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
f11
f12
f13
f14
f15
f16
f17
f18
f19
f20
f21
f22
f23
f24
f25
f26
f27
f28
f29
f30
f31
f32
f33
f34
f35
f36
f37
f38
f39
f40


f1
f2
f3
f4
f5
f6
f7
f8
f9
f10
f11
f12
f13
f14
f15
f16
f17
f18
f19
f20
f21
f22
f23
f24
f25
f26
f27
f28
CM
Malloc
f30
Free
f35
f37
f39
f31
f36
f38
f40

Evaluation

20 monolithic firmware images
(ground truth)

P²IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling

Bo Feng, *Northeastern University*
Alejandro Mera, *Northeastern University*
Long Lu, *Northeastern University*

HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation

ARTIFACT EVALUATED
usenix ASSOCIATION
PASSED

Abraham A. Clements[*,1], Eric Gustafson[*,1,2],
Tobias Scharnowski[3], Paul Grosen[2], David Fritz[1], Christopher Kruegel[2],
Giovanni Vigna[2], Saurabh Bagchi[4], and Mathias Payer[5]
[1]*Sandia National Laboratories,* [2]*UC Santa Barbara,* [3]*Ruhr-Universität Bochum,*
[4]*Purdue University,* [5]*École Polytechnique Fédérale de Lausanne*
{aacleme, djfritz}@sandia.gov, tobias.scharnowski@rub.de,
{edg, pcgrosen, chris, vigna}@cs.ucsb.edu,
sbagchi@purdue.edu, mathias.payer@epfl.ch

What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices

Toward the Analysis of Embedded Firmware through Automated Re-hosting

Eric Gustafson[1,2], Marius Muench[3], Chad Spensky[1], Nilo Redini[1], Aravind Machiry[1], Yanick Fratantonio[3]
Aurélien Francillon[3], Davide Balzarotti[3], Yung Ryn Choe[2], Christopher Kruegel[1], and Giovanni Vigna[1]
[1]*University of California, Santa Barbara*
{edg, cspensky, nredini, machiry, chris, vigna}@cs.ucsb.edu
[2]*Sandia National Laboratories*
{edgusta, yrchoe}@sandia.gov
[3]*EURECOM*
{marius.muench, francill, yanick.fratantonio, balzarot}@eurecom.fr

Evaluation

20 monolithic firmware images (ground truth)

P[2]IM: Scalable and Hardware-independent Firmware sting Automatic Peripheral Interface Modeling

Bo Feng, Northeastern University
Alejandro Mera, Northeastern University
Long, Northeastern U ersity

HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation

ARTIFAC EVALUATED usenix ASSOCIATION PASSED

Abraham A. Clements[*,1], Eric Gustafson[*,1,2],
Tobias Scharnowski[3], Paul Grosen[2], David Fritz[1], Christopher Kruegel[2],
Giovanni Vigna[2], Saurabh Bagchi[4], and Mathias Payer[5]
[1]Sandia National Laboratories, [2]UC Santa Barbara, [3]Ruhr-Universität Bochum,
[4]Purdue University, [5]École Polytechnique Fédérale de Lausanne
{aacleme, djfritz}@sandia.gov, tobias.scharnowski@rub.de,
{edg, pcgrosen, chris, vigna}@cs.ucsb.edu,
sbagchi@purdue.edu, mathias.payer@epfl.ch

u Co upt Is Not What You Crash: enges i Fuzzing Embedded Devices

the Analysis of bedded Firmware through Automated Re-hosting

Eric Gustafson[1,2], Mari uench[3], Chad Spensky[1], Nilo Redini[1], Aravind Machiry[1], Yanick Fratantonio[3]
avide Balzarotti[3], Yung Ryn Choe[2], Christopher Kruegel[1], and Giovanni Vigna[1]
[1]University of California, Santa Barbara
{edg, cspensky, nredini, machiry, chris, vigna}@cs.ucsb.edu
[2]Sandia National Laboratories
{edgusta, yrchoe}@sandia.gov
[3]EURECOM
{marius.muench, francill, yanick.fratantonio, balzarot}@eurecom.fr

Evaluation

799 monolithic firmware images
(wild dataset)

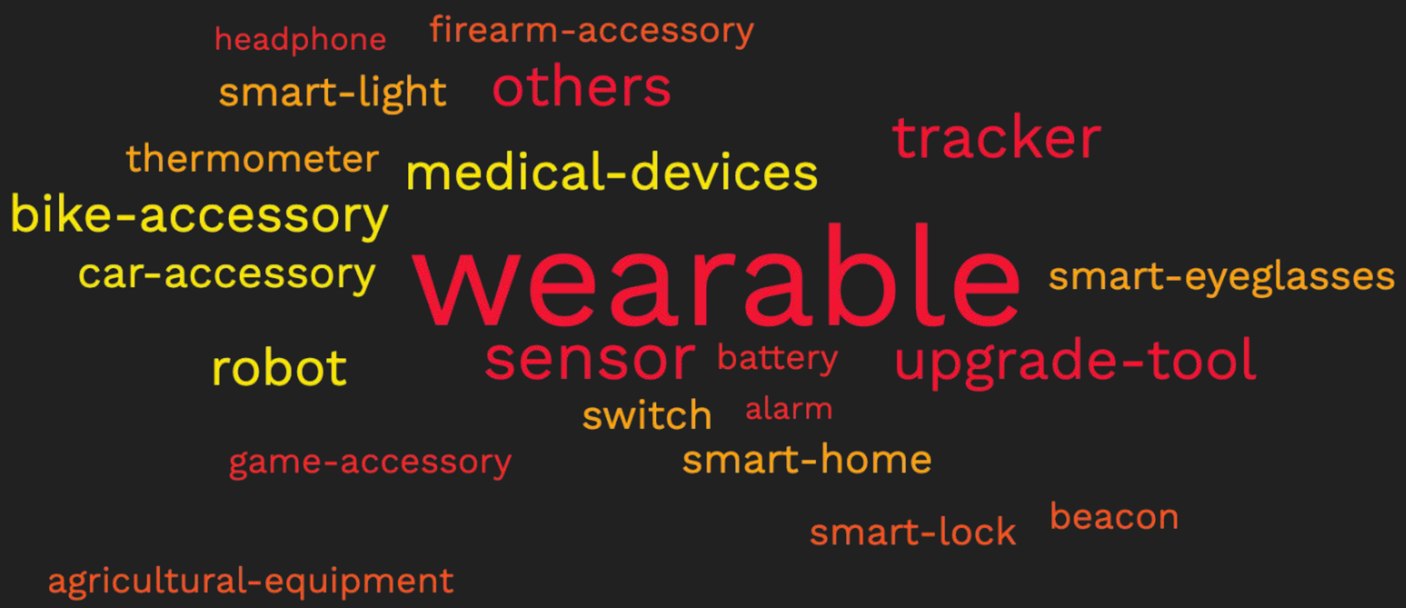
FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware

Haohuang Wen
wen.423@osu.edu
The Ohio State University

Zhiqiang Lin
zlin@cse.ohio-state.edu
The Ohio State University

Yinqian Zhang
yinqian@cse.ohio-state.edu
The Ohio State University

Evaluation

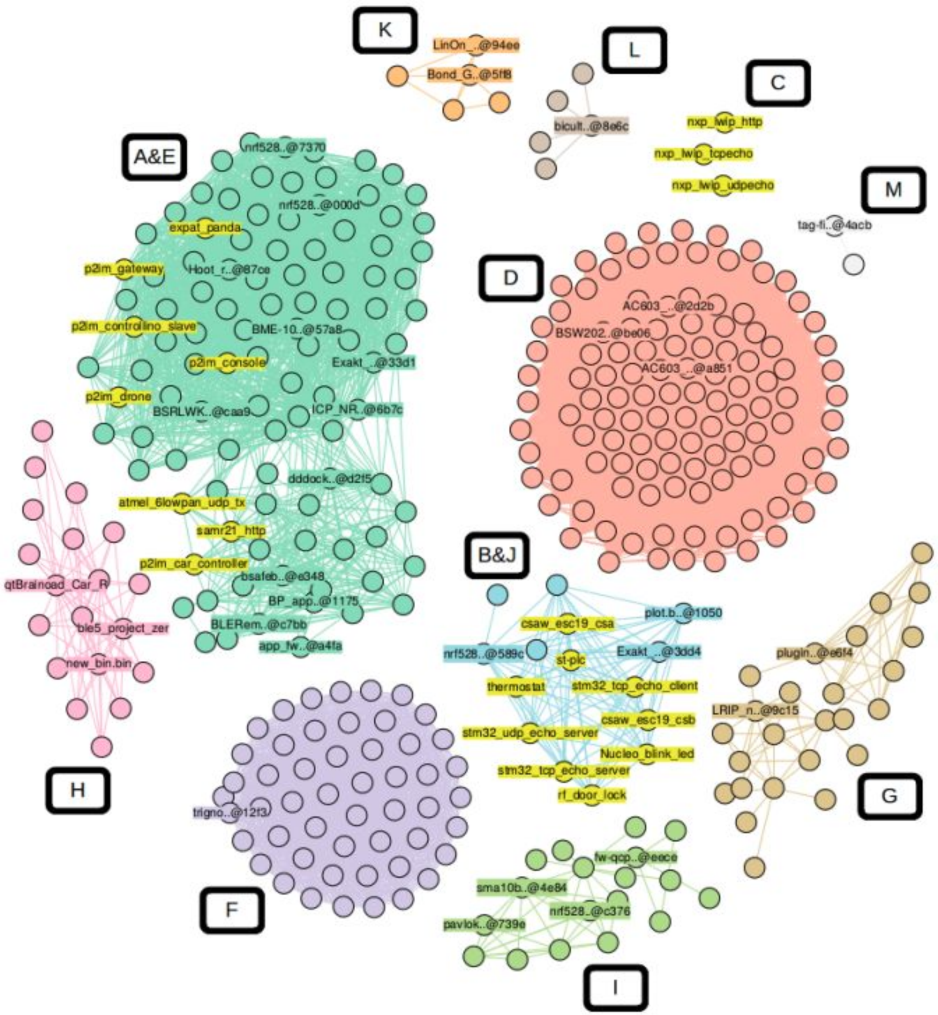

headphone
firearm-accessory
smart-light
others
tracker
thermometer
medical-devices
bike-accessory
car-accessory
wearable
smart-eyeglasses
robot
sensor
battery
upgrade-tool
switch
alarm
game-accessory
smart-home
smart-lock
beacon
agricultural-equipment

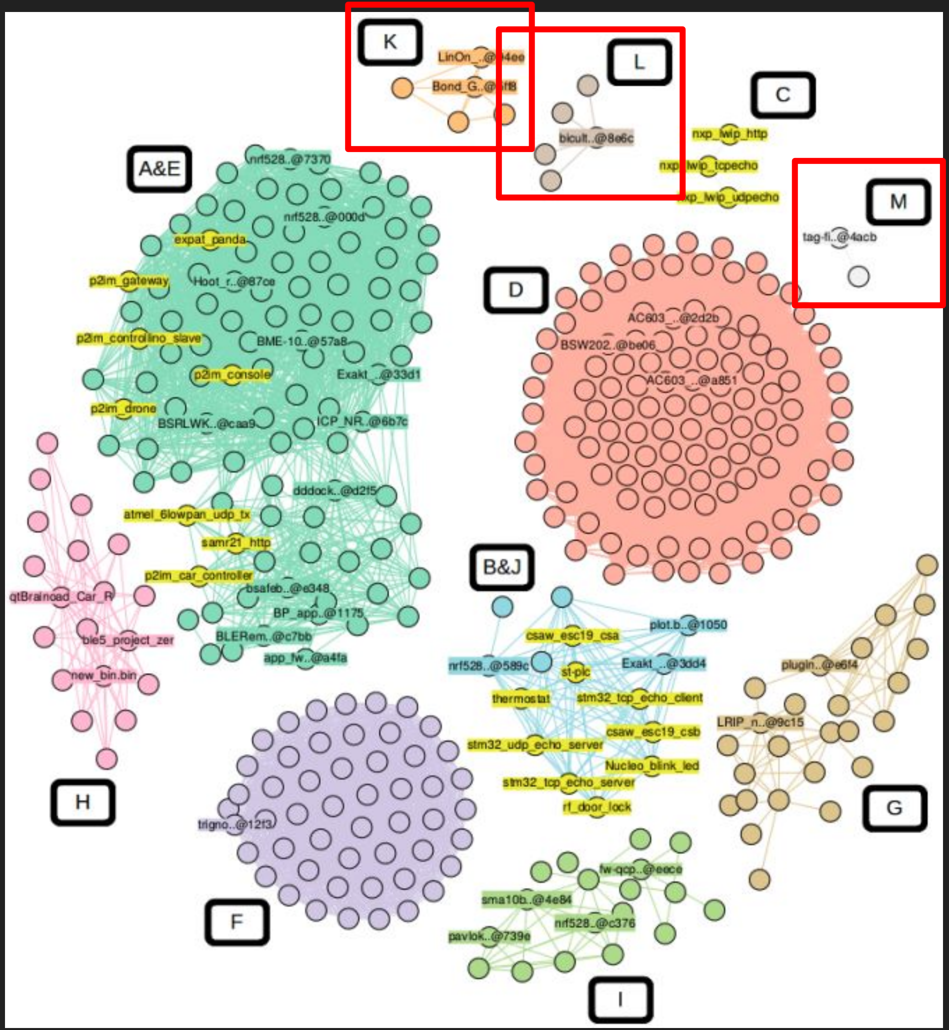
Evaluation

- 799 monolithic firmware images
- 340 use a dynamic memory allocator (~42%)

Evaluation

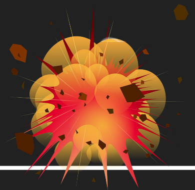
- 799 monolithic firmware images
- 340 use a dynamic memory allocator (~42%)
- 10 different HML families in 32 different variations


K
L
C
A&E
M
D
B&J
H
F
G
I


K
L
C
M
A&E
D
B&J
H
F
G
I
LinOn_..@94ee
Bond_G..@5ff8
bicult..@8e6c
nxp_lwip_http
nxp_lwip_tcpecho
nxp_lwip_udpecho
tag-fi..@4acb
nrf528..@7370
nrf528..@000d
expat_panda
p2im_gateway
Hoot_r..@87ce
p2im_controllino_slave
BME-10..@57a8
p2im_console
Exakt_..@33d1
p2im_drone
BSRLWK..@caa9
ICP_NR..@6b7c
dddock..@d2f5
atmel_6lowpan_udp_tx
samr21_http
p2im_car_controller
bsafeb..@e348
BP_app..@1175
BLERem..@c7bb
app_fw..@a4fa
qtBrainoad_Car_R
ble5_project_zer
new_bin.bin
AC603_..@2d2b
BSW202..@be06
AC603_..@a851
plot.b..@1050
csaw_esc19_csa
nrf528..@589c
Exakt_..@3dd4
st-plc
thermostat
stm32_tcp_echo_client
csaw_esc19_csb
stm32_udp_echo_server
Nucleo_blink_led
stm32_tcp_echo_server
rf_door_lock
plugin..@e6f4
LRIP_n..@9c15
trigno..@12f3
fw-qcp..@eece
sma10b..@4e84
nrf528..@c376
pavlok..@739e


SECURITY TESTING
imgflip.com

HML Security Testing



Heap exploitation primitive

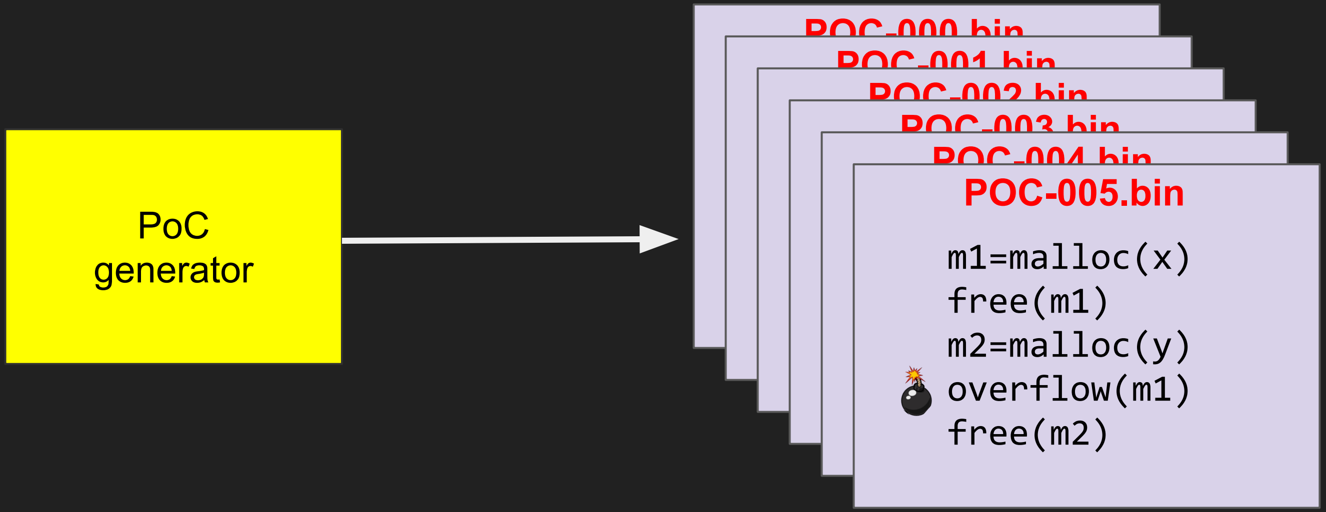
- Heap overflow
- Use-after-free
- Double-free
- Fake-free

Heap vulnerable state

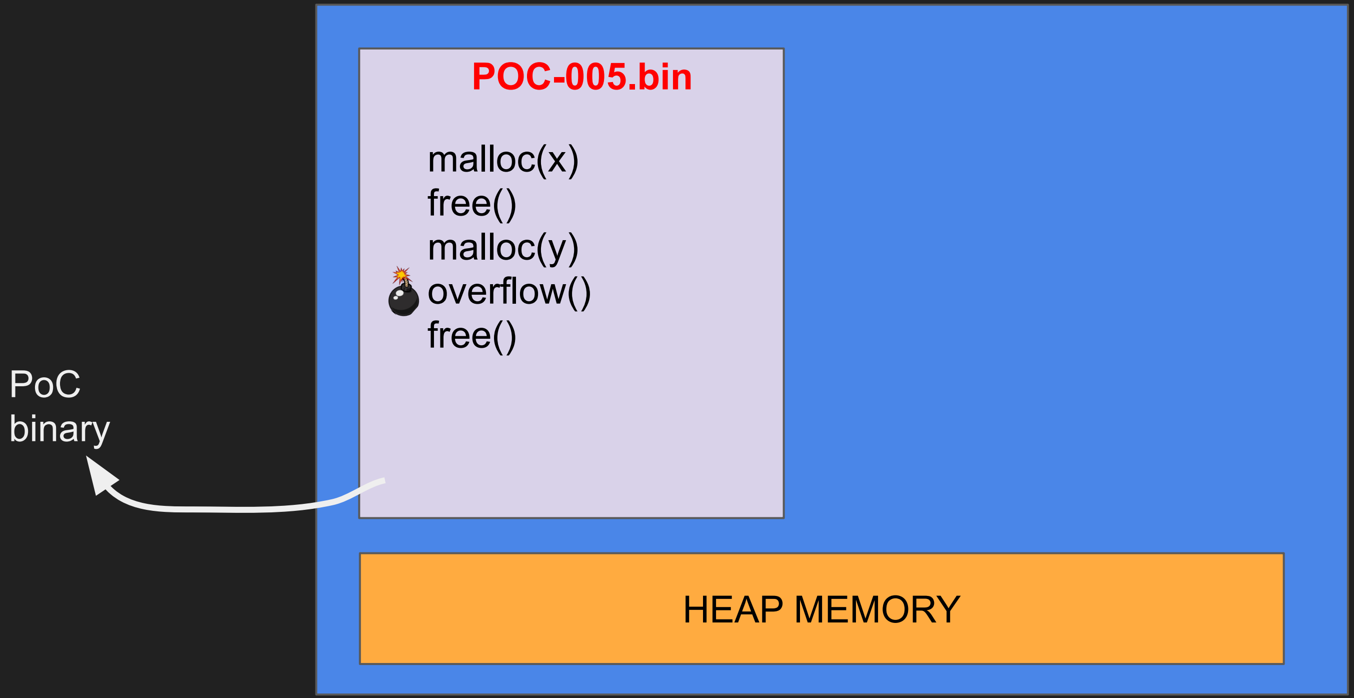
- Overlapped chunk
- Out-of-heap allocation
- Restricted write
- Arbitrary write

HML Security Testing

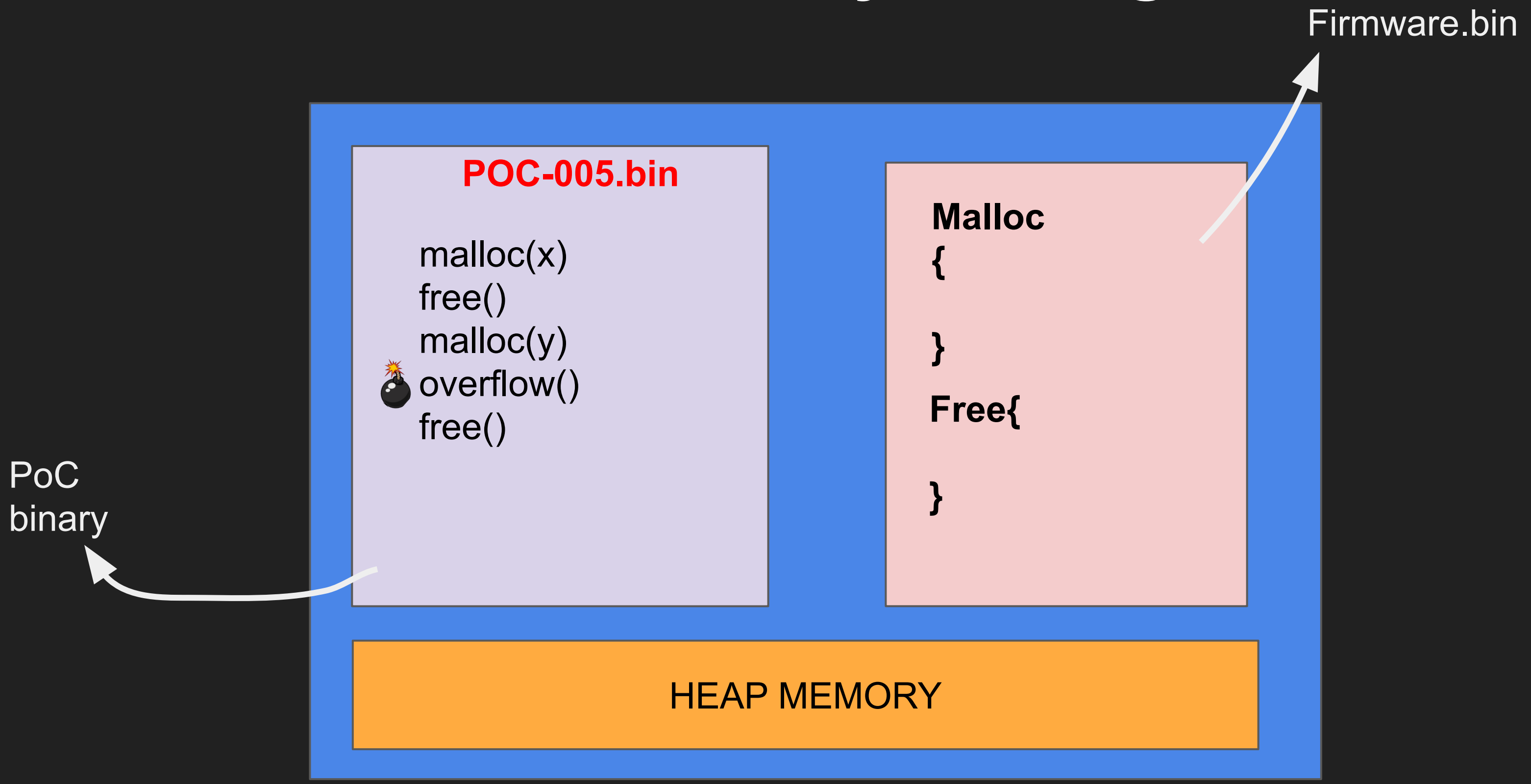
- Generation of PoC(s) that will be symbolically traced



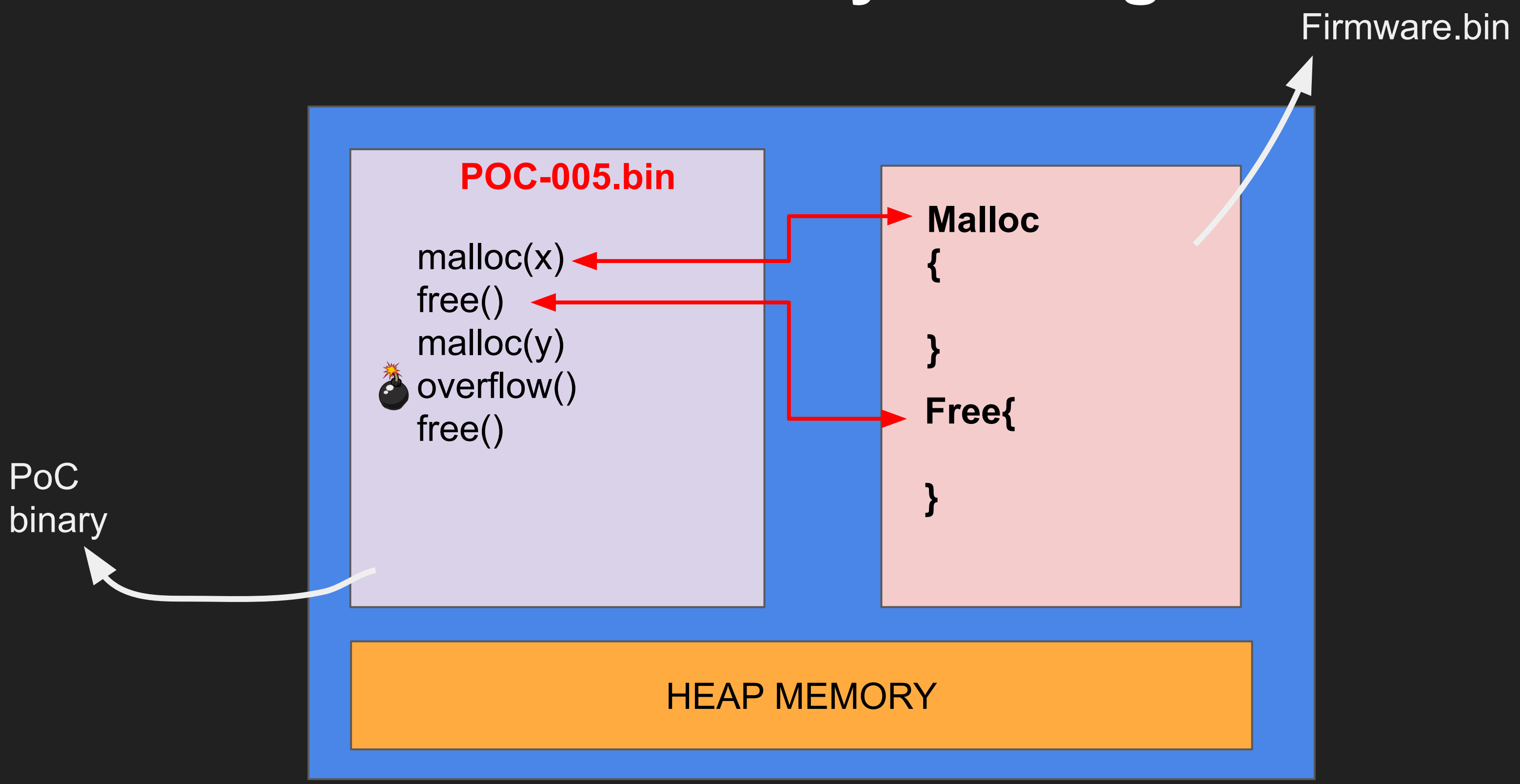
HML Security Testing



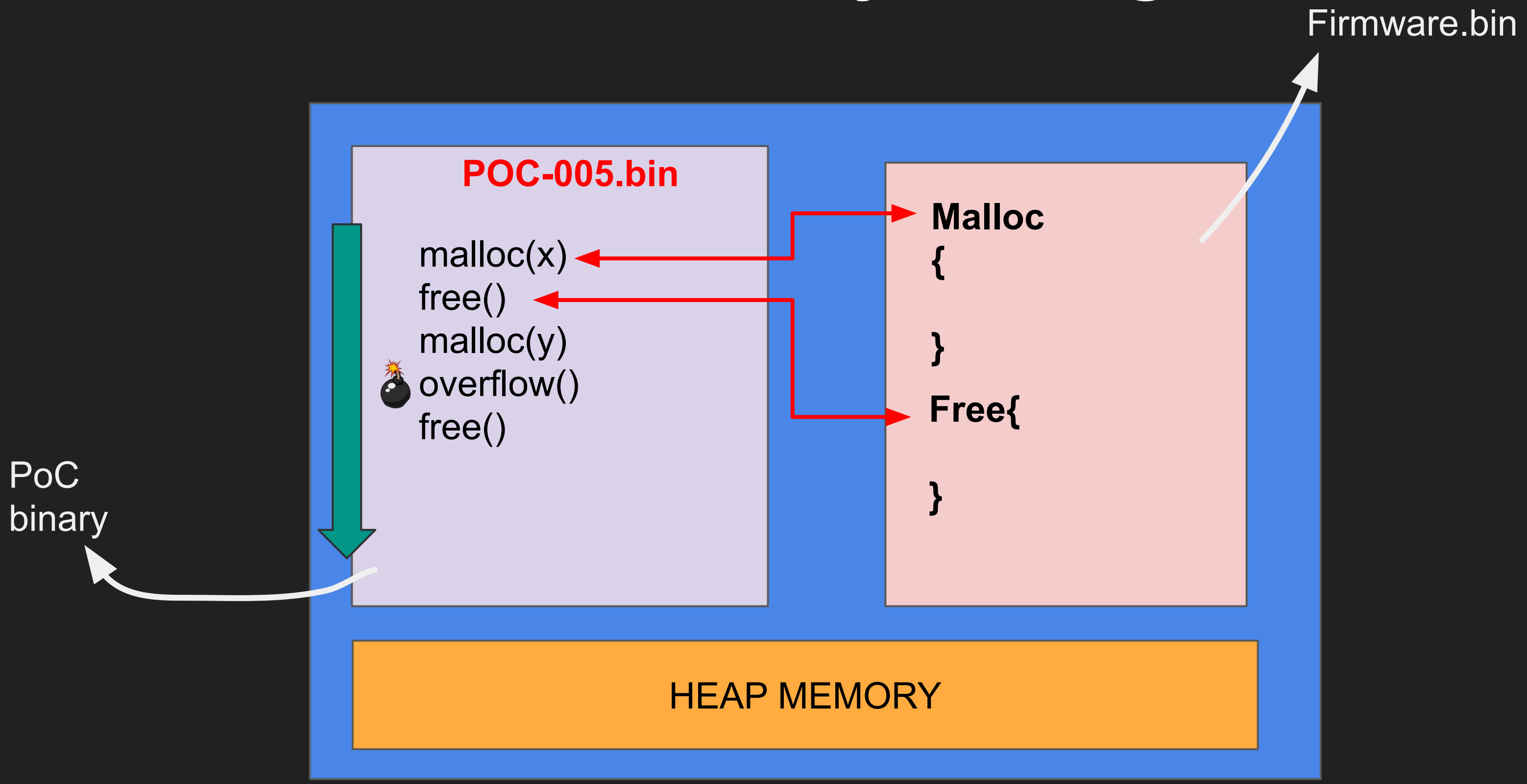
HML Security Testing



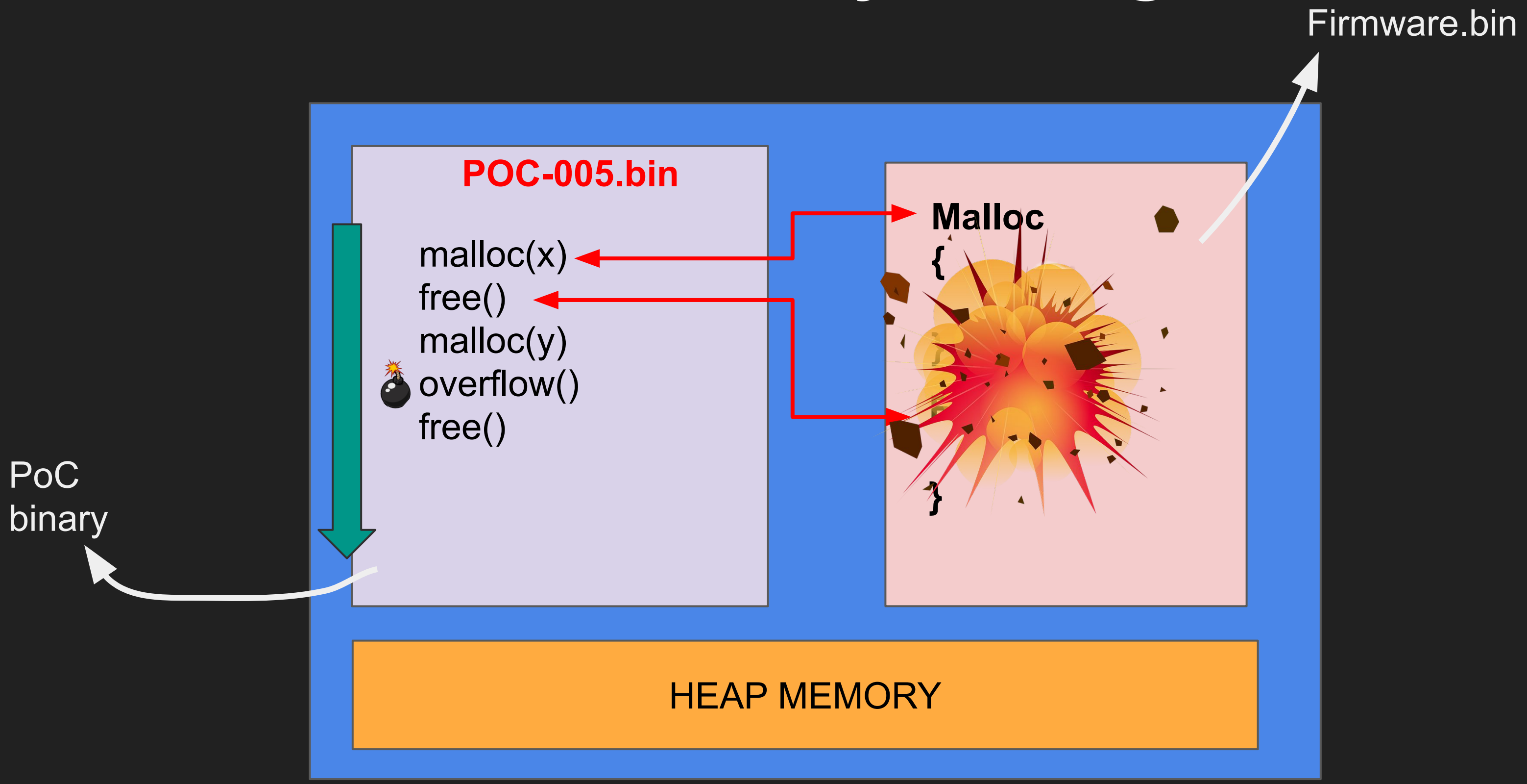
HML Security Testing

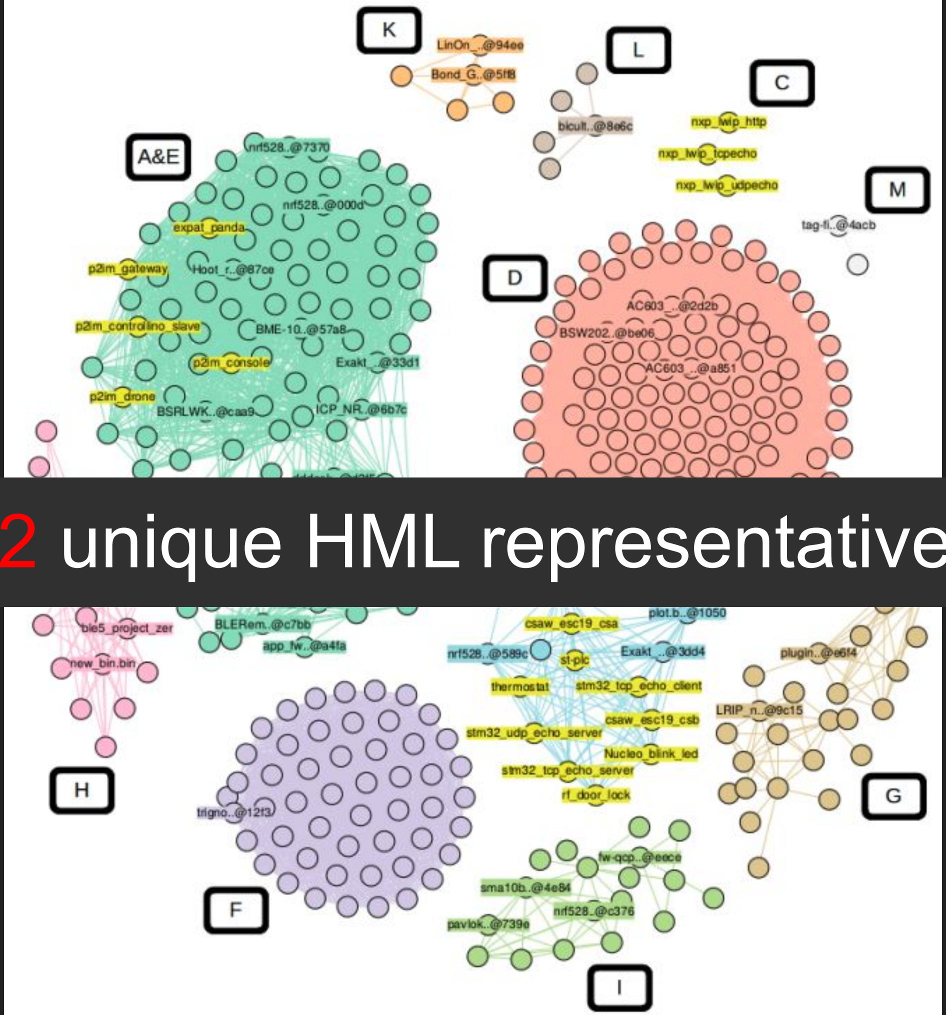


HML Security Testing



HML Security Testing

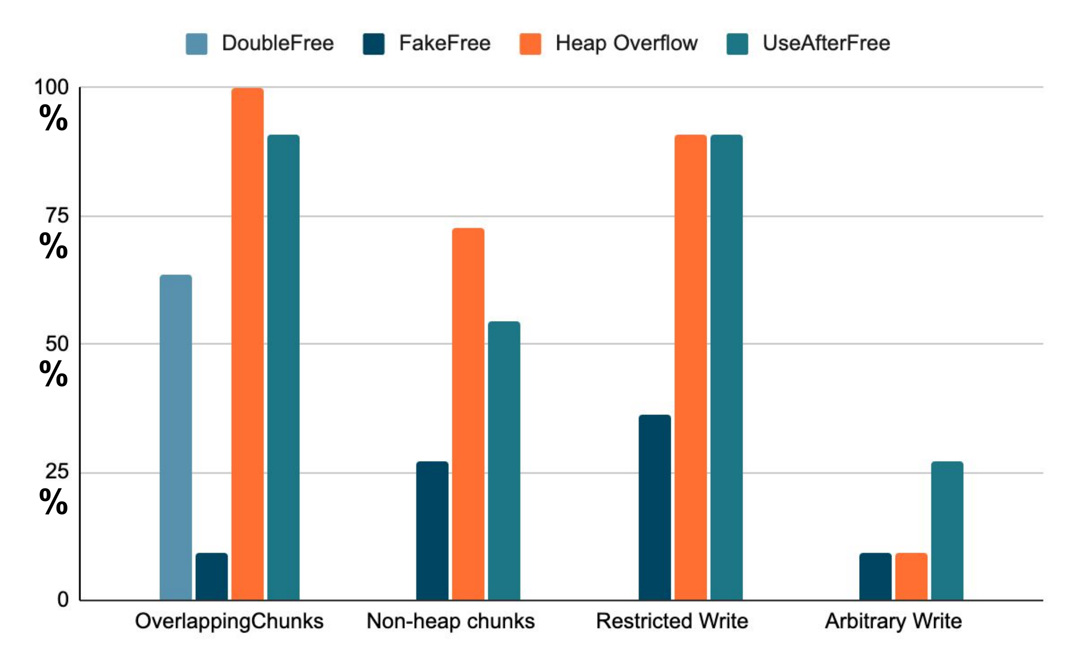




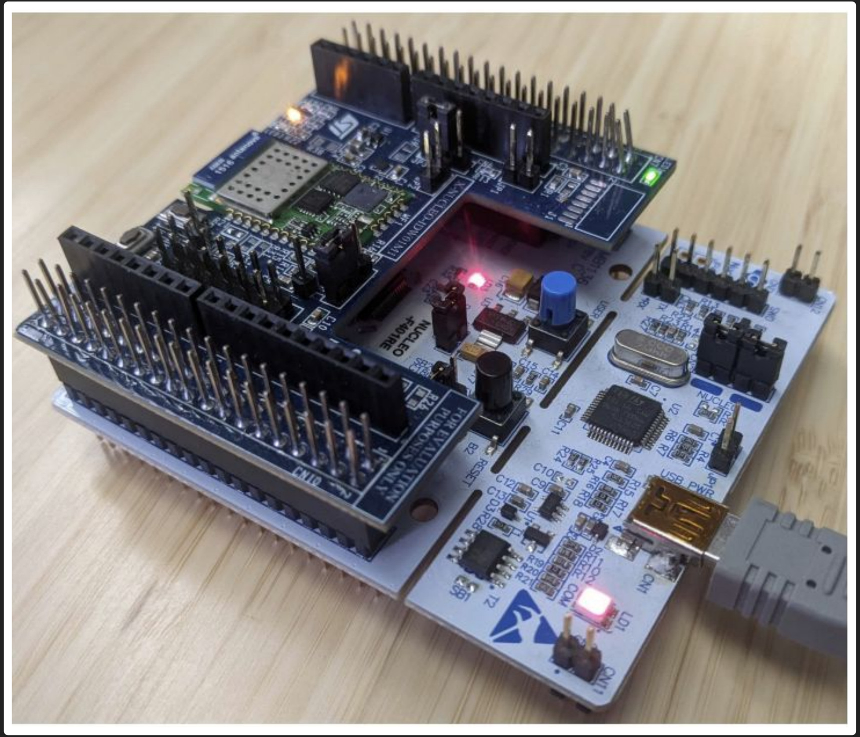
32 unique HML representatives

Security Test Results

- *All* the tested HML were vulnerable to at least one heap exploitation primitive

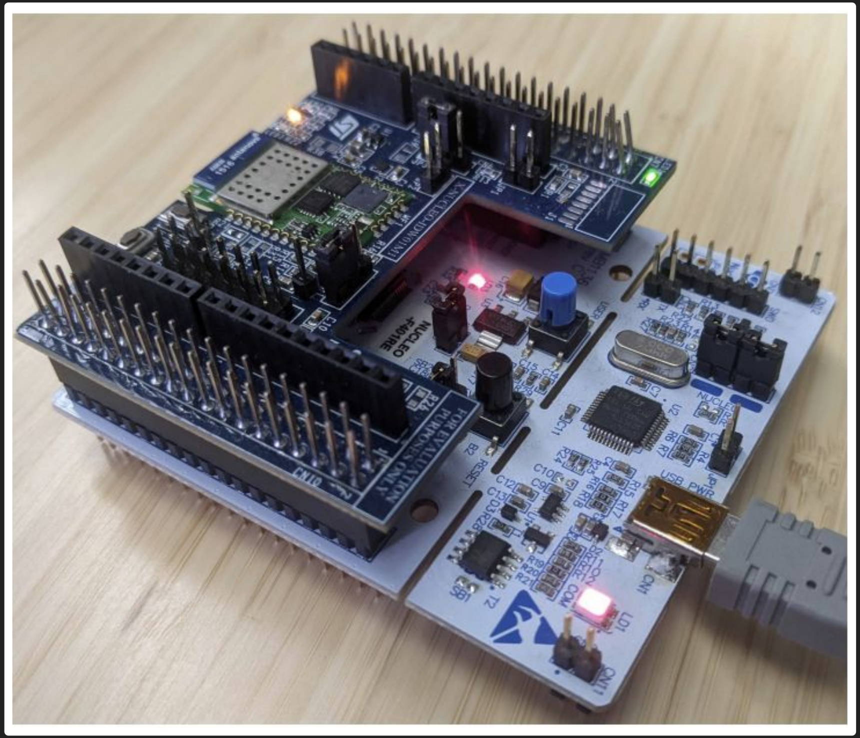


Hardware Example



Developed application that uses malloc/free

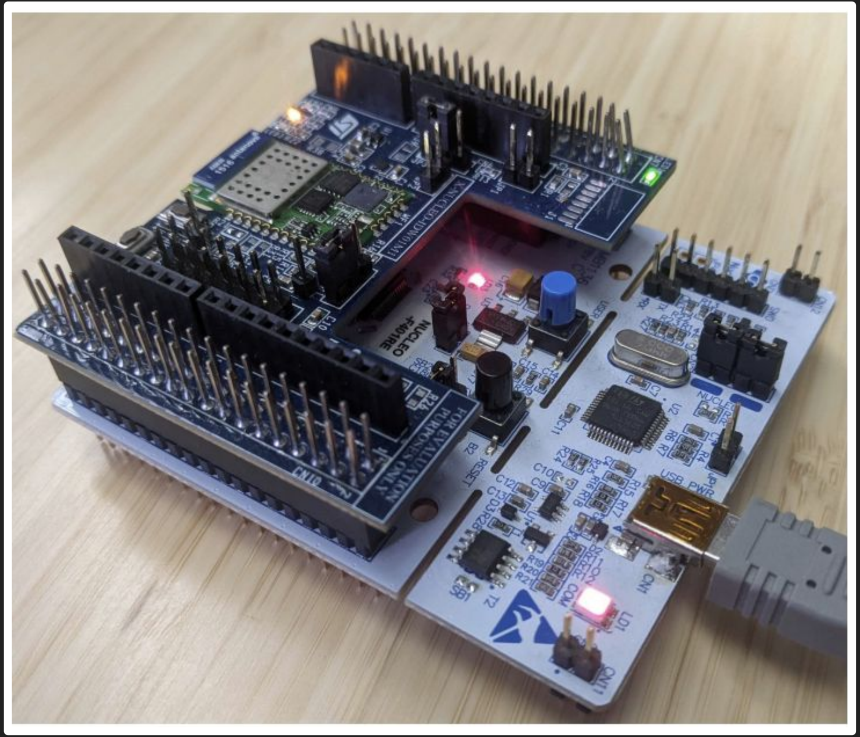
Hardware Example



Developed application that uses malloc/free

Unknown HML included in the firmware

Hardware Example

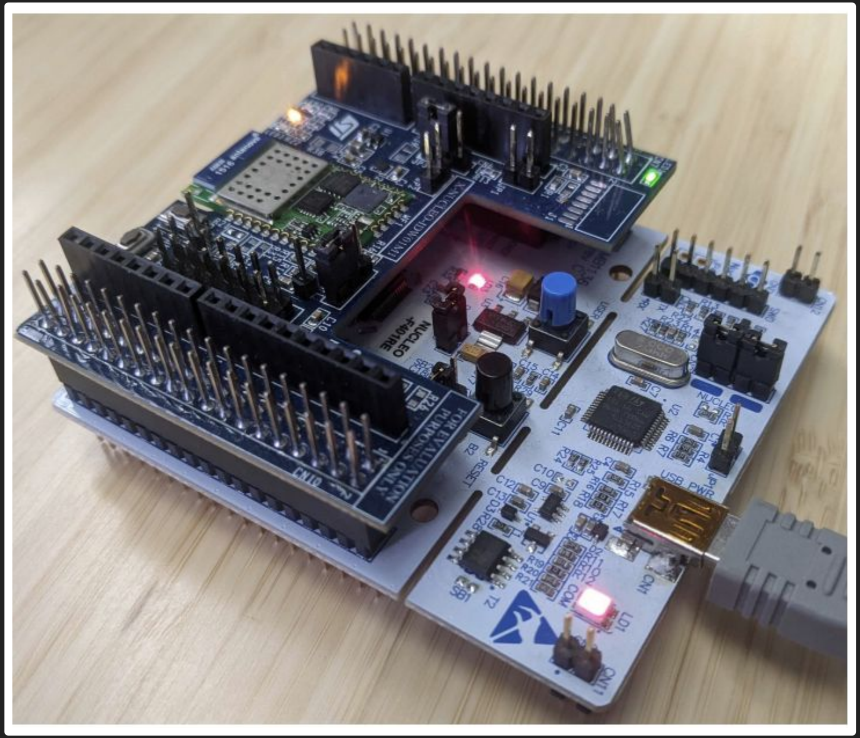


Developed application that uses malloc/free

Unknown HML included in the firmware

Heapster detected possible attacks

Hardware Example



Developed application that uses malloc/free

Unknown HML included in the firmware

Heapster detected possible attacks

Attacks confirmed on the board

Conclusions

Takeaways

- Dynamic memory allocators are often used in monolithic firmware

Takeaways

- Dynamic memory allocators are often used in monolithic firmware
- Different and unique implementations in different variants

Takeaways

- Dynamic memory allocators are often used in monolithic firmware
- Different and unique implementations in different variants
- *Every* tested HML was vulnerable to at least one heap exploitation technique

- Open source
  - github.com/ucsb-seclab/heapster
- Support
  - https://angr.io/invite/
  - Ping me @degrigis

Thanks!

---

degrigis@ucsb.edu

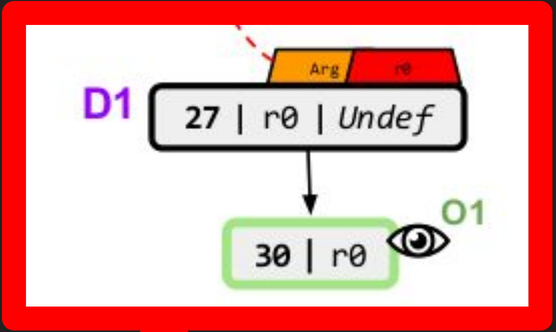
@degrigis

Extra

Static Taint Engine

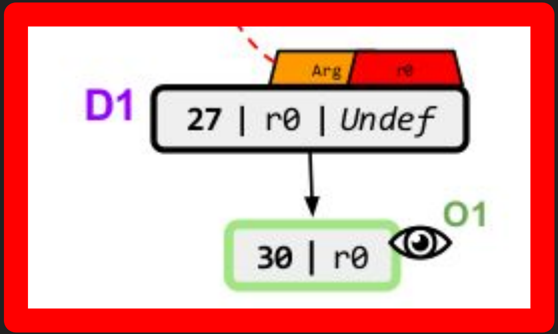
```
last = 0x0
void mem_init(){
 last = 0x2000C000;
}

int malloc(x){
 chunk = last
 last = last + x
 return chunk;
}

int baz(){
 void *x = 0x2000;
 return x;
}

void bar(x,y){
 if(y==0)
   v1 = baz();
 else
   v1 = malloc(y);
 foo(v1);
}

void foo(a){
  int b[10];
  memcmp(a, b, 10);
}
```

```
malloc:
 mov r0, <arg_0>
 ldr r1, [last]
 add r0, r0, last
 str r0, [last]
 mov r0, r1
 ret ;[O4]

baz:
 mov r0, 0x2000
 ret ;[O3]

bar:
 mov r0, <arg0>
 mov r1, <arg1>
 cmp r1, 0
 bne tag
 call baz
 b return
tag:
 call malloc
return:
 call foo ;[O2]
 ret

foo:
 mov r0, <arg0>
 mov r1, var_b
 mov r2, 0x10
 call memcmp ;[O1]
 ret
```



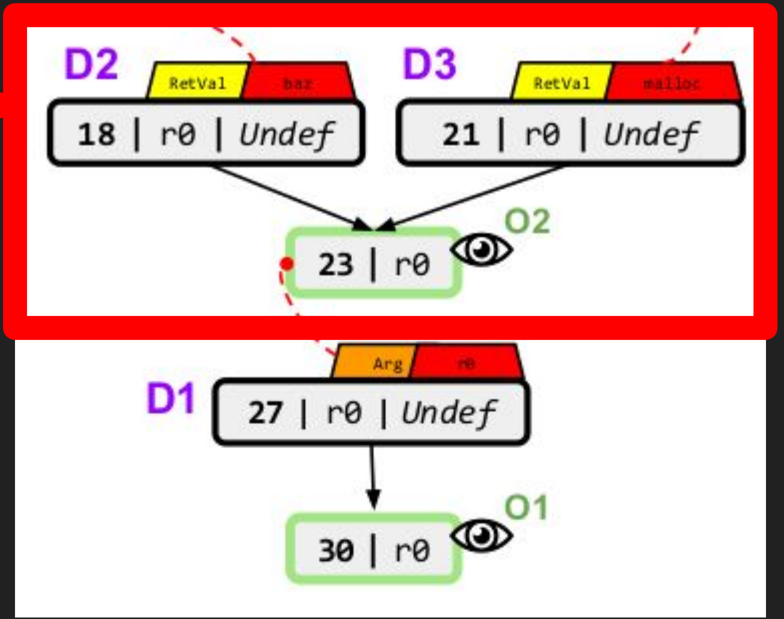
```
last = 0x0
void mem_init(){
 last = 0x2000C000;
}

int malloc(x){
 chunk = last
 last = last + x
 return chunk;
}

int baz(){
 void *x = 0x2000;
 return x;
}

void bar(x,y){
 if(y==0)
   v1 = baz();
 else
   v1 = malloc(y);
 foo(v1);
}

void foo(a){
  int b[10];
  memcmp(a, b, 10);
}
```

```
malloc:
 mov r0, <arg_0>
 ldr r1, [last]
 add r0, r0, last
 str r0, [last]
 mov r0, r1
 ret ;[O4]

baz:
 mov r0, 0x2000
 ret ;[O3]

bar:
 mov r0, <arg0>
 mov r1, <arg1>
 cmp r1, 0
 bne tag
 call baz
 b return
tag:
 call malloc
return:
 call foo ;[O2]
 ret

foo:
 mov r0, <arg0>
 mov r1, var_b
 mov r2, 0x10
 call memcmp ;[O1]
 ret
```



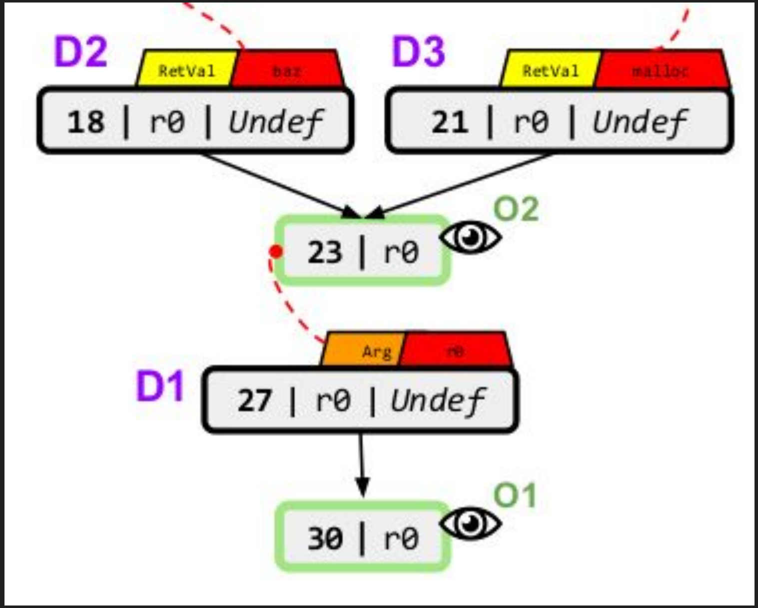
```
last = 0x0
void mem_init(){
 last = 0x2000C000;
}

int malloc(x){
 chunk = last
 last = last + x
 return chunk;
}

int baz(){
 void *x = 0x2000;
 return x;
}

void bar(x,y){
 if(y==0)
   v1 = baz();
 else
   v1 = malloc(y);
 foo(v1);
}

void foo(a){
  int b[10];
  memcmp(a, b, 10);
}
```

```
malloc:
 mov r0, <arg_0>
 ldr r1, [last]
 add r0, r0, last
 str r0, [last]
 mov r0, r1
 ret ;[O4]

baz:
 mov r0, 0x2000
 ret ;[O3]

bar:
 mov r0, <arg0>
 mov r1, <arg1>
 cmp r1, 0
 bne tag
 call baz
 b return
tag:
 call malloc
return:
 call foo ;[O2]
 ret

foo:
 mov r0, <arg0>
 mov r1, var_b
 mov r2, 0x10
 call memcmp ;[O1]
 ret
```



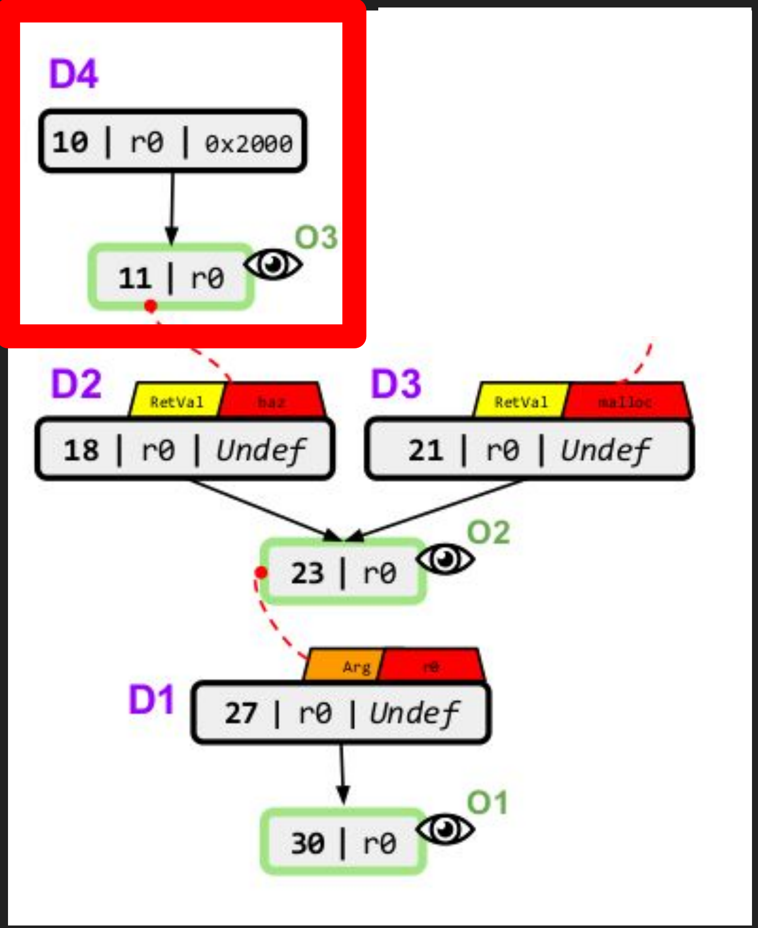
```c
last = 0x0
void mem_init(){
 last = 0x2000C000;
}

int malloc(x){
 chunk = last
 last = last + x
 return chunk;
}

int baz(){
 void *x = 0x2000;
 return x;
}

void bar(x,y){
 if(y==0)
   v1 = baz();
 else
   v1 = malloc(y);
 foo(v1);
}

void foo(a){
  int b[10];
  memcmp(a, b, 10);
}
```

```
1  malloc:
2   mov r0, <arg_0>
3   ldr r1, [last]
4   add r0, r0, last
5   str r0, [last]
6   mov r0, r1
7   ret ;[O4]
8
9  baz:
10  mov r0, 0x2000
11  ret ;[O3]
12
13 bar:
14  mov r0, <arg0>
15  mov r1, <arg1>
16  cmp r1, 0
17  bne tag
18  call baz
19  b return
20 tag:
21  call malloc
22 return:
23  call foo ;[O2]
24  ret
25
26 foo:
27  mov r0, <arg0>
28  mov r1, var_b
29  mov r2, 0x10
30  call memcmp ;[O1]
31  ret
```



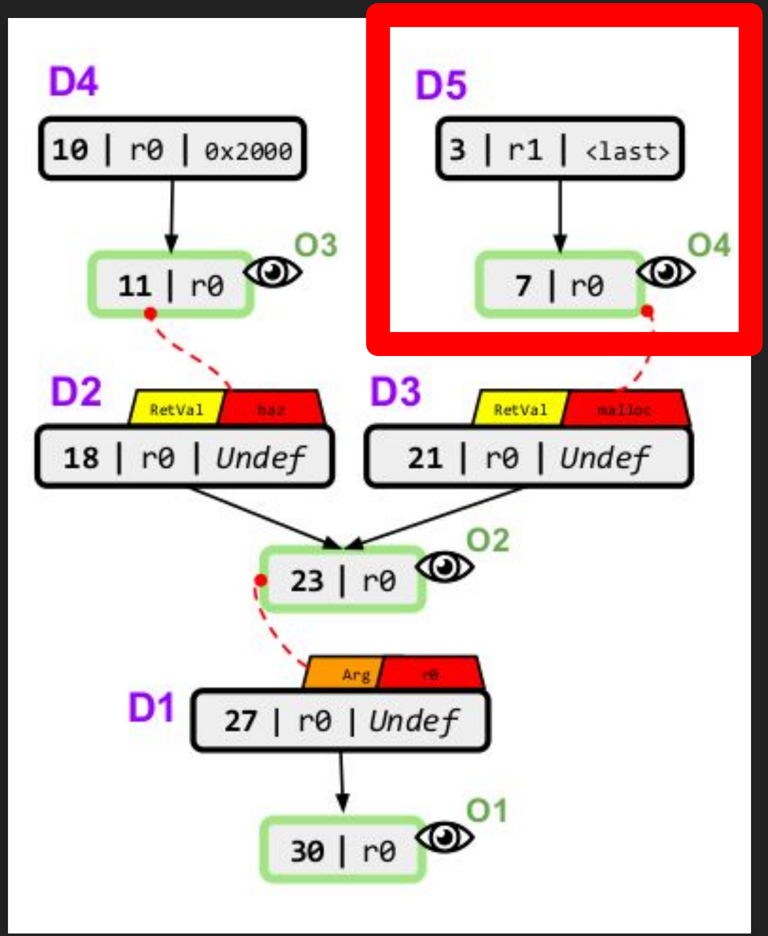
```
last = 0x0
void mem_init(){
 last = 0x2000C000;
}

int malloc(x){
 chunk = last
 last = last + x
 return chunk;
}

int baz(){
 void *x = 0x2000;
 return x;
}

void bar(x,y){
 if(y==0)
   v1 = baz();
 else
   v1 = malloc(y);
 foo(v1);
}

void foo(a){
  int b[10];
  memcmp(a, b, 10);
}
```

```
malloc:
 mov r0, <arg_0>
 ldr r1, [last]
 add r0, r0, last
 str r0, [last]
 mov r0, r1
 ret ;[O4]

baz:
 mov r0, 0x2000
 ret ;[O3]

bar:
 mov r0, <arg0>
 mov r1, <arg1>
 cmp r1, 0
 bne tag
 call baz
 b return
tag:
 call malloc
return:
 call foo ;[O2]
 ret

foo:
 mov r0, <arg0>
 mov r1, var_b
 mov r2, 0x10
 call memcmp ;[O1]
 ret
```



```c
last = 0x0
void mem_init(){
 last = 0x2000C000;
}

int malloc(x){
 chunk = last
 last = last + x
 return chunk;
}

int baz(){
 void *x = 0x2000;
 return x;
}

void bar(x,y){
 if(y==0)
   v1 = baz();
 else
   v1 = malloc(y);
 foo(v1);
}

void foo(a){
  int b[10];
  memcmp(a, b, 10);
}
```

```asm
malloc:
 mov r0, <arg_0>
 ldr r1, [last]
 add r0, r0, last
 str r0, [last]
 mov r0, r1
 ret ;[O4]

baz:
 mov r0, 0x2000
 ret ;[O3]

bar:
 mov r0, <arg0>
 mov r1, <arg1>
 cmp r1, 0
 bne tag
 call baz
 b return
tag:
 call malloc
return:
 call foo ;[O2]
 ret

foo:
 mov r0, <arg0>
 mov r1, var_b
 mov r2, 0x10
 call memcmp ;[O1]
 ret
```



Firmware Initialization

Firmware initialization (Cortex-M)

- **Reset Handler execution**
  - Need to execute the compiler-injected stub that unpacks .bss and global data

```
void __noreturn Reset_Handler()
{
  int i; // r1
  int v1; // r0

  for ( i = 0; (unsigned int)&sdata + i < 0x200009CC; i += 4 )
    *(void **)((char *)&sdata + i) = *(void **)(i + 134248564);
  v1 = SystemInit();
  start(v1);
  while ( 1 )
    ;
}
```

Firmware initialization

- **Heap Auxiliary Functions**
  - Responsible to write heap-specific data in memory

```
void mem_init()
{
  unsigned int v0; // [sp+4h] [bp+4h]

  ram = (unsigned int)&unk_1FFF3B57 & 0xFFFFFFFC;// stores 0x1fff3b54 in 0x1fff9368
  v0 = (unsigned int)&unk_1FFF3B57 & 0xFFFFFFFC;
  *(_WORD *)v0 = 0x5800;
  *(_WORD *)(v0 + 2) = 0;
  *(_BYTE *)(v0 + 4) = 0;
  ram_end = ram + 0x5800;                    // stores 0x1fff9354 in 0x1fff936c
  *(_BYTE *)(ram + 0x5804) = 1;
  *(_WORD *)ram_end = 0x5800;
  *(_WORD *)(ram_end + 2) = 0x5800;
  lfree = ram;                               // stores 0x1fff3b54 in 1fff9370
}
```

Firmware initialization

- Can be implemented in many ways...

```c
int heap_init()
{
  memset_alt(byte_200050B8, 0x610u, 'H');
  memset_alt(byte_200056C8, 0xA20u, 'V');
  memset_alt(byte_200060E8, 0x1040u, 'L');
  memset_alt(byte_20007128, 0x1D18u, 'M');
  memset_alt(byte_20008E40, 0x1CB0u, 'm');
  memset_alt(byte_2000AAF0, 0xE10u, 's');
  memset_alt(byte_2000B900, 0x41A0u, 'T');
  memset_alt(byte_2000FAA0, 0xD20u, 't');
  bzero((int)word_20005068, 48);
  init_mem(0, (int)byte_200050B8);
  init_mem(1, (int)byte_200056C8);
  init_mem(2, (int)byte_200060E8);
  init_mem(3, (int)byte_20007128);
  init_mem(4, (int)byte_20008E40);
  init_mem(5, (int)byte_2000AAF0);
  init_mem(6, (int)byte_2000B900);
  return init_mem(7, (int)byte_2000FAA0);
}
```

pacemaker

```c
unsigned int __fastcall init_mem(int a1, int dest_addr)
{
  int v2; // r6
  char *v3; // r4
  int v4; // r5
  unsigned int result; // r0
  int v6; // r3

  v2 = 5 * a1;
  v3 = (char *)&dword_8085D76 + 10 * a1;
  v4 = *((unsigned __int16 *)v3 + 2);
  byte_20005098[a1] = dest_addr;
  for ( result = 0; *((unsigned __int16 *)v3 + 4) > result; ++result )
  {
    *(_WORD *)(dest_addr + 4) = *((_WORD *)&dword_8085D76 + v2);
    *(_WORD *)(dest_addr + 6) = 0;
    *(_WORD *)(dest_addr + 8) = 0;
    *(_BYTE *)(dest_addr + 10) = 0;
    *(_BYTE *)(dest_addr + 11) = 0;
    if ( *((unsigned __int16 *)v3 + 4) - 1 == result )
      v6 = 0;
    else
      v6 = dest_addr + v4;
    *(_DWORD *)dest_addr = v6;
    dest_addr += v4;
  }
  return result;
}
```

Firmware initialization

- Can be implemented in many ways...

```
int sub_11EFC()
{
  _BYTE v1[6]; // [sp+0h] [bp-D0h] BYREF
  __int16 v2; // [sp+6h] [bp-CAh] BYREF
  __int16 v3; // [sp+Ah] [bp-C6h] BYREF
  __int16 v4; // [sp+Eh] [bp-C2h] BYREF
  __int16 v5; // [sp+12h] [bp-BEh] BYREF
  __int16 v6; // [sp+2Ah] [bp-A6h] BYREF
  __int16 v7; // [sp+2Eh] [bp-A2h] BYREF
  __int16 v8; // [sp+32h] [bp-9Eh] BYREF
  __int16 v9; // [sp+36h] [bp-9Ah] BYREF

  sub_11440(119);
  sub_3B4(v1, &byte_2000288D, 6);
  sub_3B4(&v2, &unk_20004370, 4);
  sub_3B4(&v3, &unk_20004378, 4);
  sub_3B4(&v4, &unk_2000437C, 4);
  sub_402(&v5, 24);
  sub_3B4(&v6, &dword_200028A8, 4);
  sub_3B4(&v7, &dword_200028AC, 4);
  sub_3B4(&v8, &dword_200028A4, 4);
  sub_3B4(&v9, &unk_20004382, 96);
  return sub_114EC(487424, v1, 150);
}
```

Calls free() to insert first Free chunk.

AC603_VIITA

HML Classification

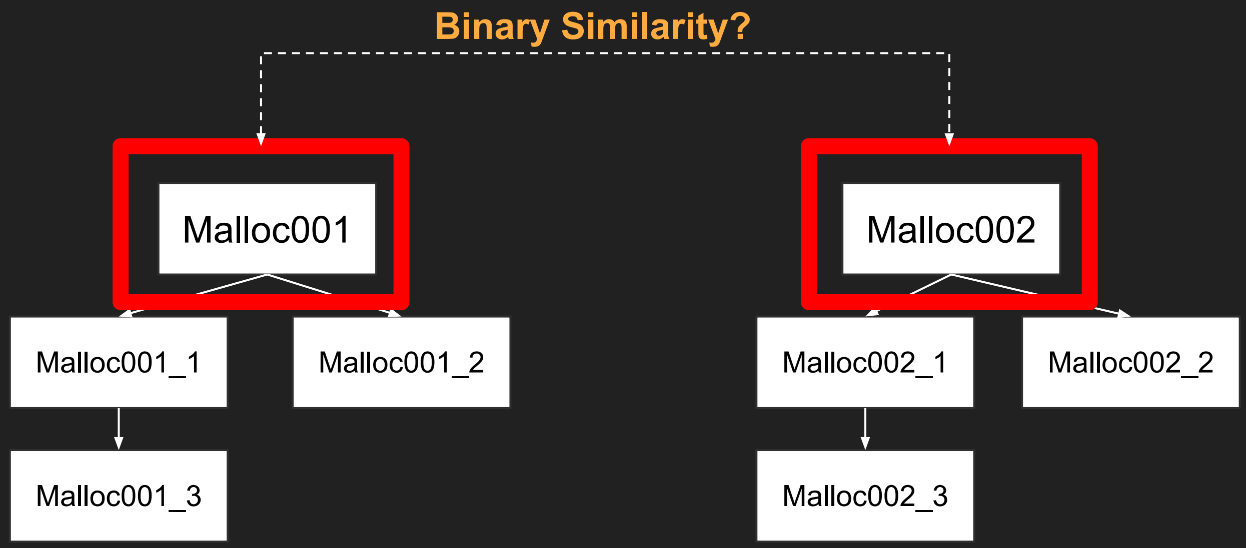
HML Classifications

Malloc001

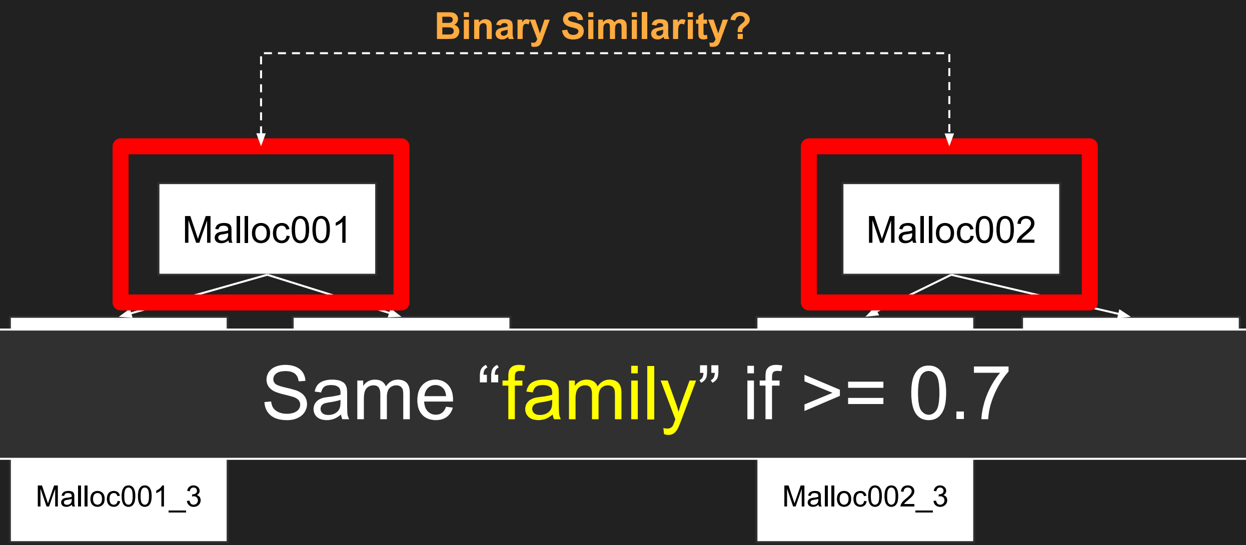
Malloc002

HML Classifications

HML Classifications

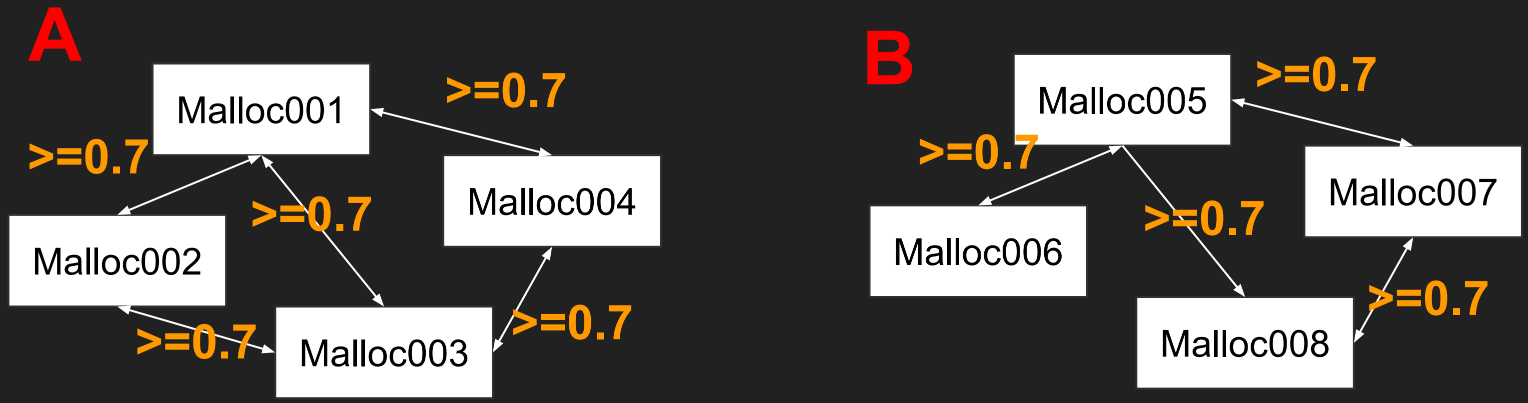


HML Classifications



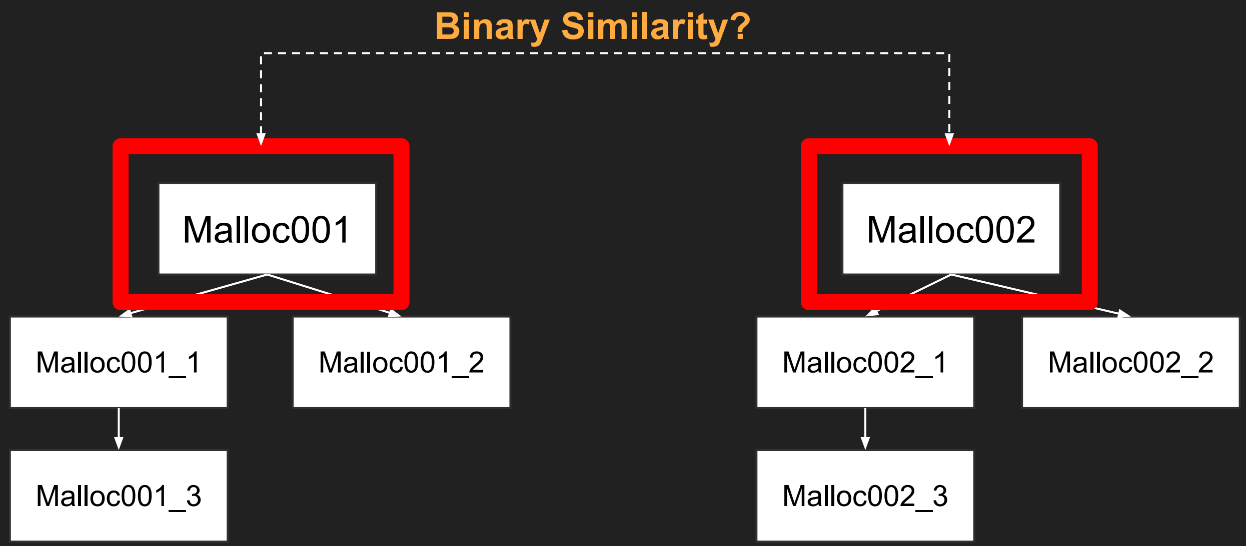
Same “family” if >= 0.7

HML Classifications

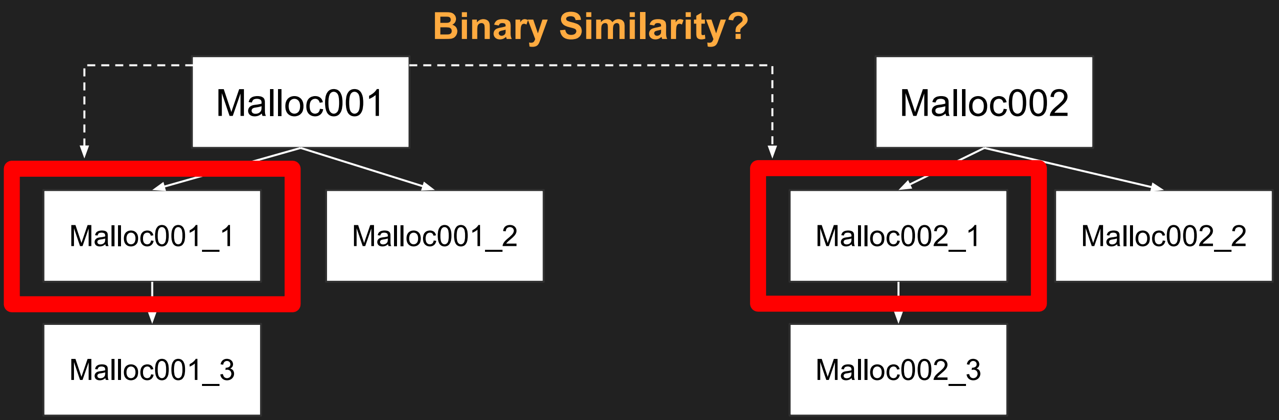


HML Classifications

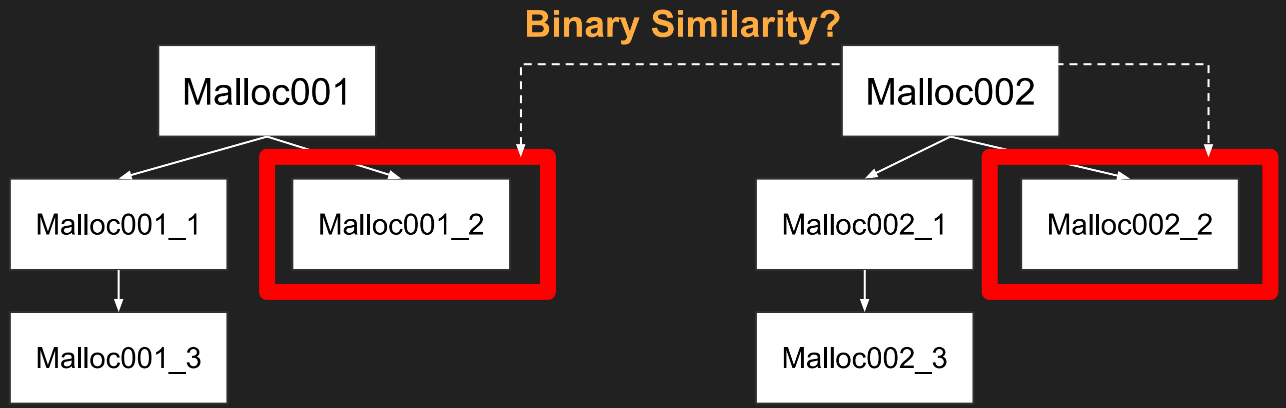
HML Classifications



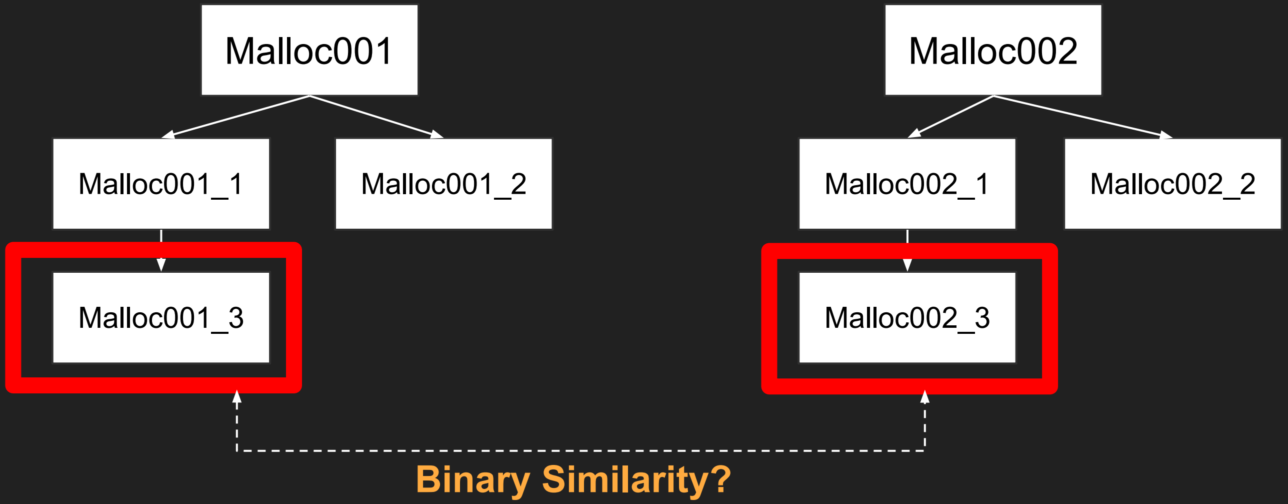
HML Classifications



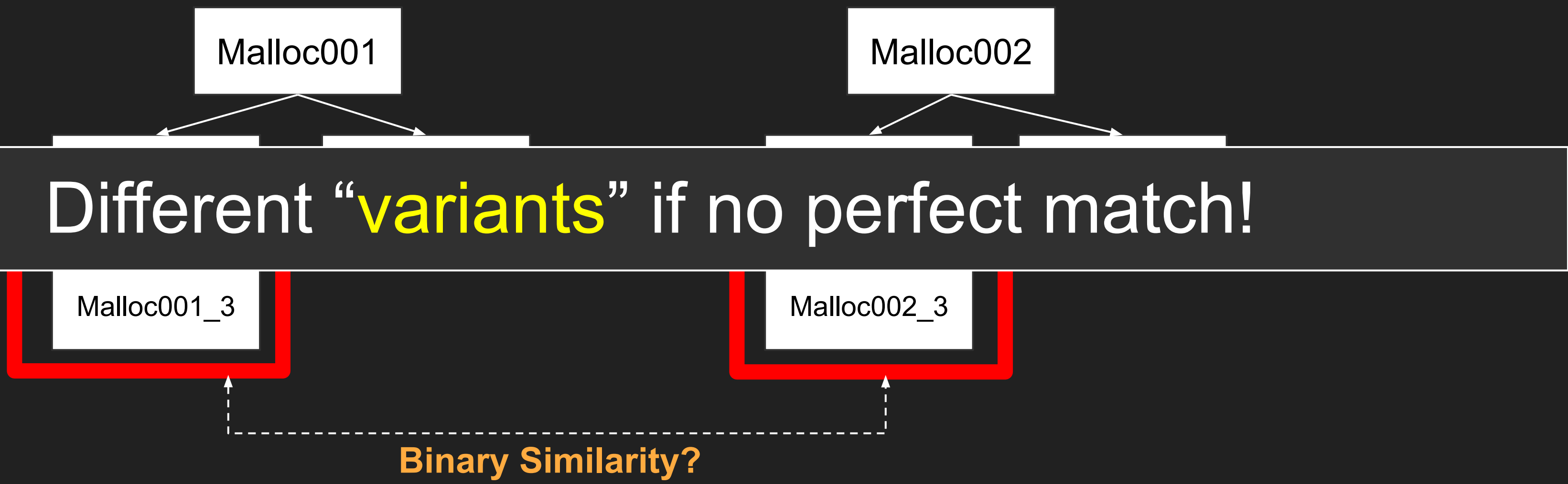
HML Classifications



HML Classifications

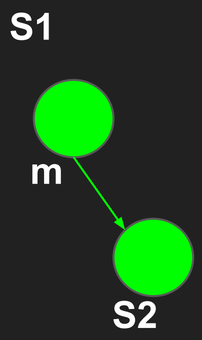


HML Classifications

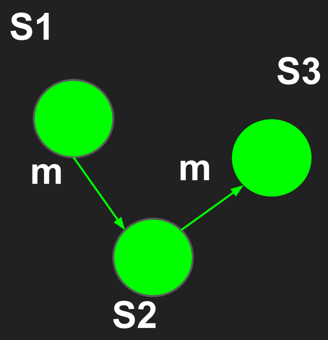


PoC Tracing

HML Security Testing
(PoC tracing)

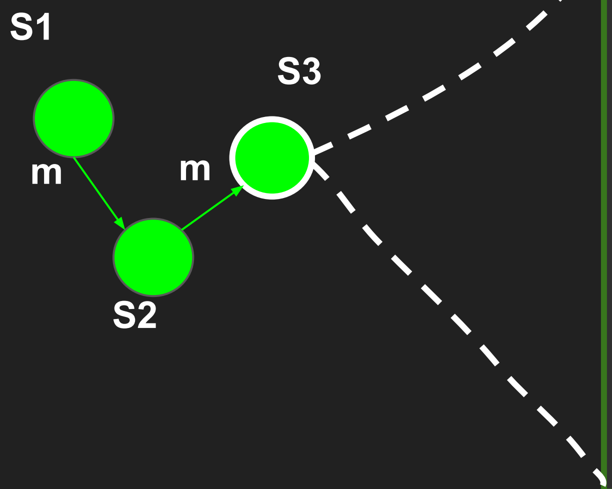


HML Security Testing
(PoC tracing)



HML Security Testing

(PoC tracing)

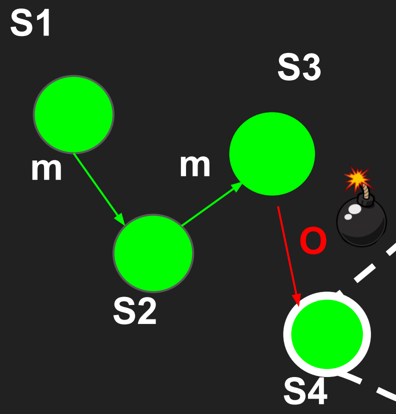


Heap memory

0x20000000	0x00010000
0x20000004	0x41414141
0x20000008	0x41414141
0x2000000c	0x41414141
0x20000010	0x41414141
0x20000014	0x00000000
0x20000018	0x00001010
0x2000001c	0x41414141
0x20000020	0x41414141
0x20000024	0x41414141
0x20000028	0x41414141
0x2000002c	0x00551230
0x20000030	0x00000000
0x20000034	0x00000000
0x20000038	0x00000000
0x2000003c	0x00000000

HML Security Testing

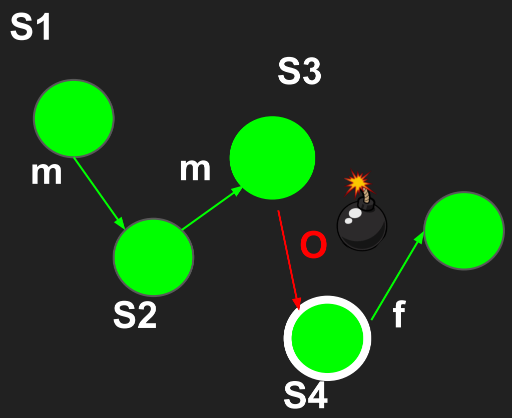
(PoC tracing)



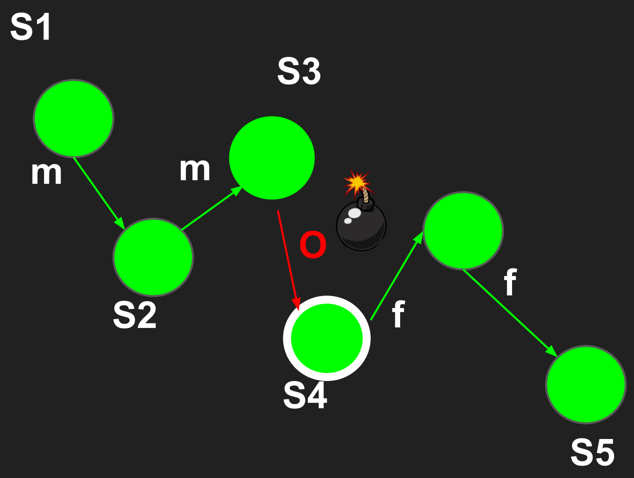
Heap memory

0x20000000	symb__varA
0x20000004	symb__varB
0x20000008	symb__varC
0x2000000c	0x41414141
0x20000010	0x41414141
0x20000014	0x00000000
0x20000018	0x00001010
0x2000001c	0x41414141
0x20000020	0x41414141
0x20000024	0x41414141
0x20000028	0x41414141
0x2000002c	0x00551230
0x20000030	0x00000000
0x20000034	0x00000000
0x20000038	0x00000000
0x2000003c	0x00000000

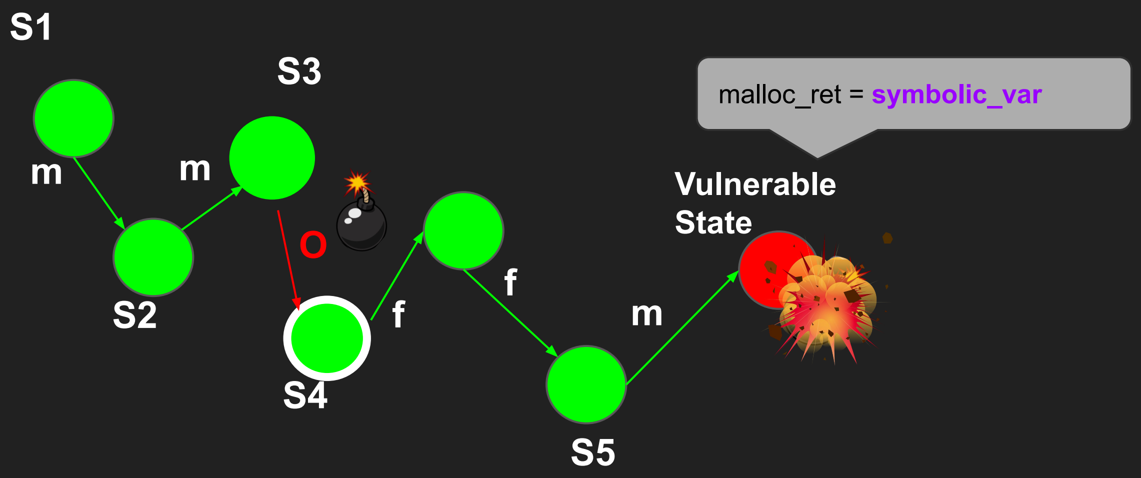
HML Security Testing
(PoC tracing)



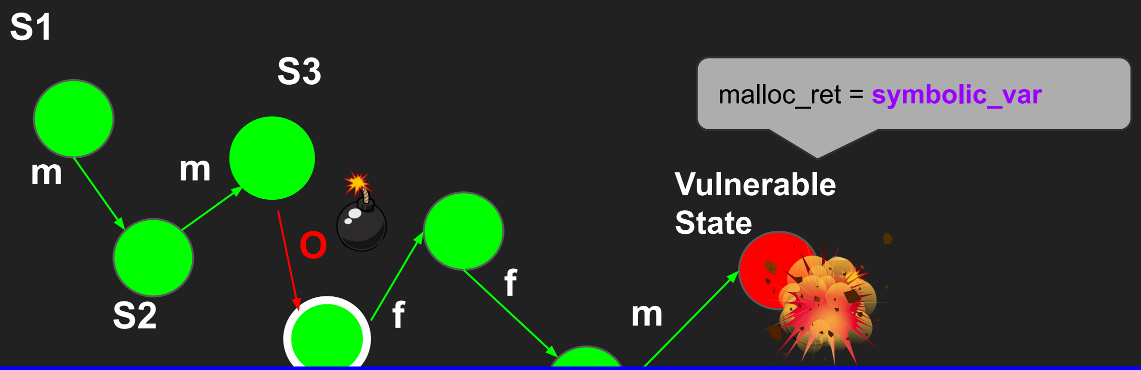
HML Security Testing
(PoC tracing)



HML Security Testing
(PoC tracing)

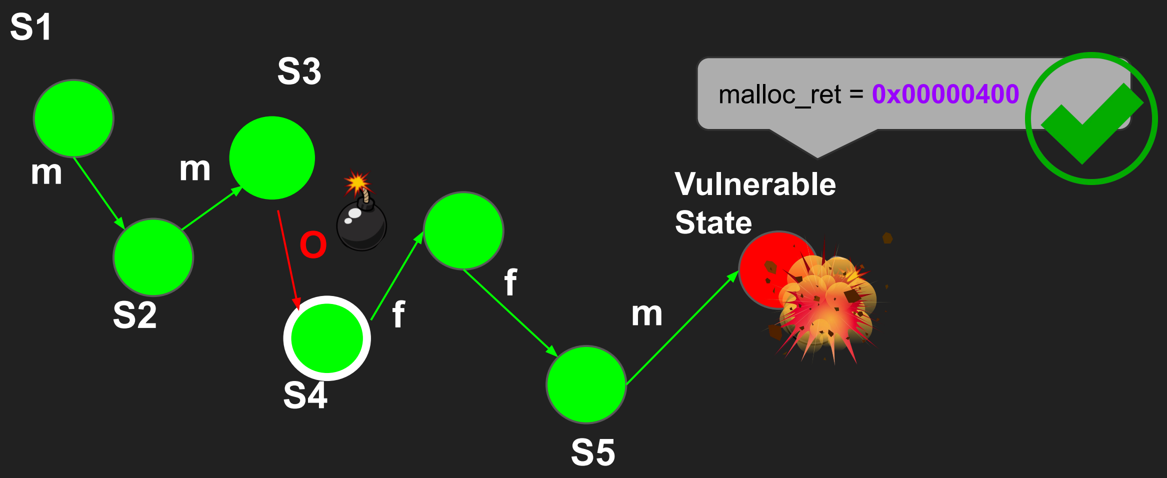


HML Security Testing
(PoC tracing)

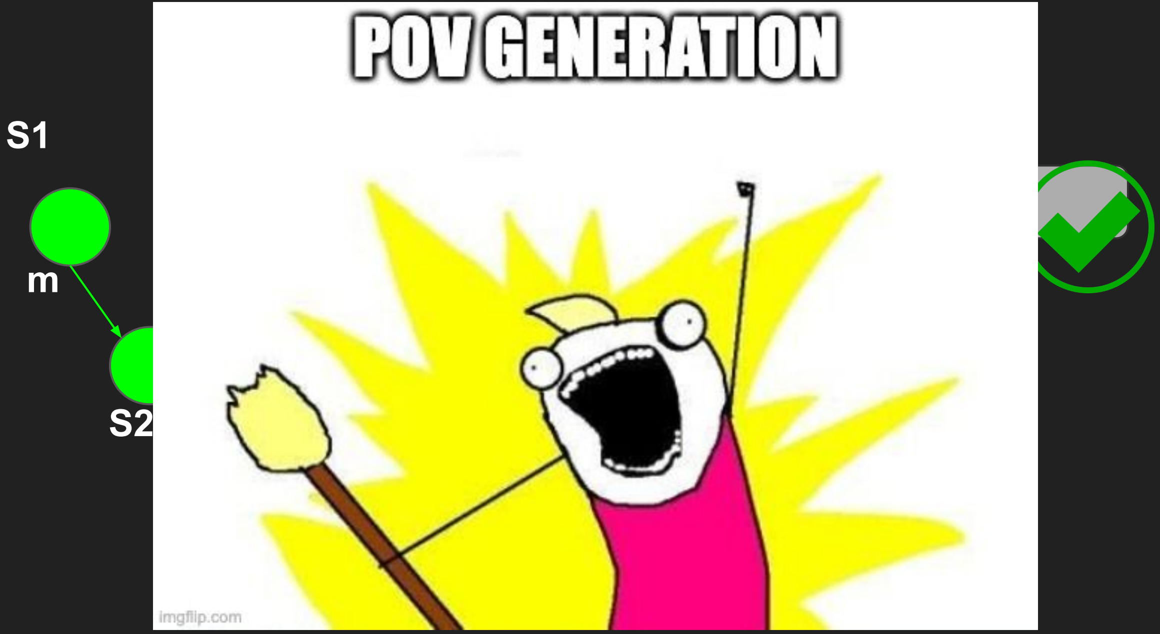


Can I concretize value to be out-of-heap?

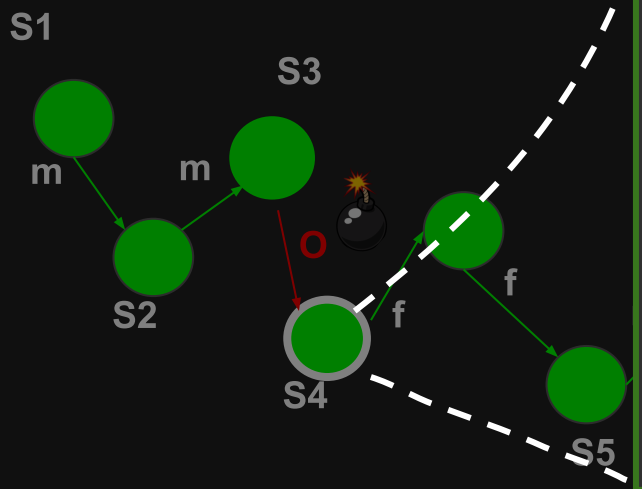
HML Security Testing
(PoC tracing)



HML Security Testing
(PoC tracing)

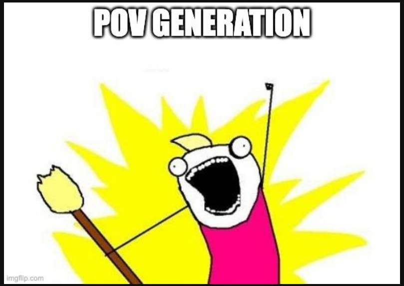


HML Security Testing
(PoV generation)

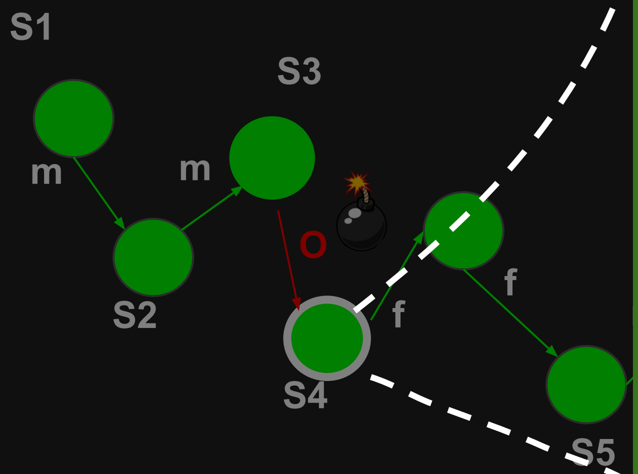


Heap memory

0x20000000	symb__varA
0x20000004	symb__varB
0x20000008	symb__varC
0x2000000c	0x41414141
0x20000010	0x41414141
0x20000014	0x00000000
0x20000018	0x00001010
0x2000001c	0x41414141
0x20000020	0x41414141
0x20000024	0x41414141
0x20000028	0x41414141
0x2000002c	0x00551230
0x20000030	0x00000000
0x20000034	0x00000000
0x20000038	0x00000000
0x2000003c	0x00000000

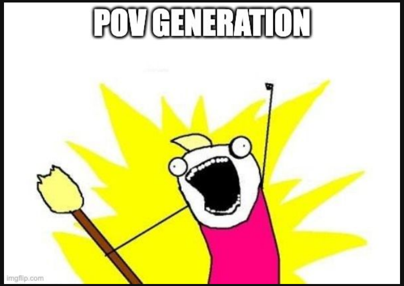


HML Security Testing
(PoV generation)



Heap memory

0x20000000	0x000000400
0x20000004	0x00000000c
0x20000008	0x000000100
0x2000000c	0x41414141
0x20000010	0x41414141
0x20000014	0x00000000
0x20000018	0x00001010
0x2000001c	0x41414141
0x20000020	0x41414141
0x20000024	0x41414141
0x20000028	0x41414141
0x2000002c	0x00551230
0x20000030	0x00000000
0x20000034	0x00000000
0x20000038	0x00000000
0x2000003c	0x00000000



```
m1=malloc(X)
free(m1)
m2=malloc(Y)
overflow(m1)
free(m2)
```

POC-005.bin

```
m1=malloc(10)
free(m1)
m2=malloc(20)
overflow(m1)
free(m2)
```

POV-005.bin

PoV Feasibility

Security Test Results

What about the feasibility of PoVs attacks?

Security Test Results

What abou[...] of PoVs attacks?

```
m1=malloc(10)
free(m1)
m2=malloc(20)
overflow(m1)
free(m2)
```

POV-005.bin

Security Test Results

```
m1=malloc(10)
free(m1)
m2=malloc(20)
overflow(m1)
free(m2)
```

Security Test Results

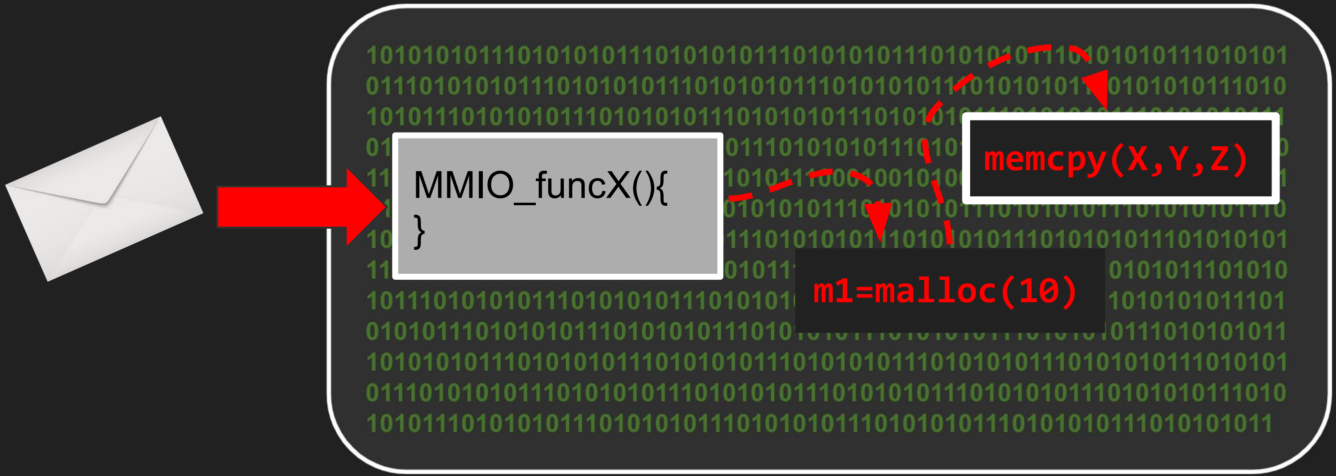
```
m1=malloc(10)
free(m1)
m2=malloc(20)
overflow(m1)
free(m2)
```

No re-hosting, extremely challenging…

Security Test Results

P1: Can I reach a call to malloc from functions that read data from MMIO regions?

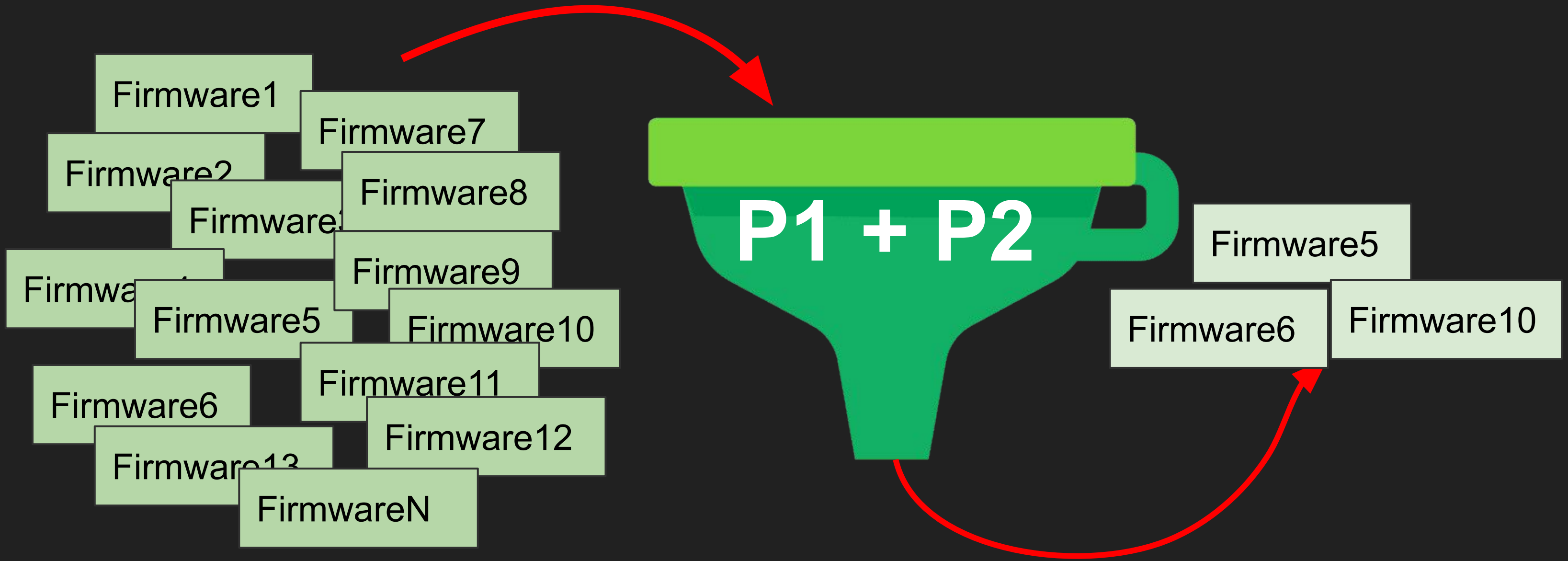
Security Test Results



P2: Can I observe a flow from
malloc to a memcpy with not constant size?

Security Test Results

- Manually review of pre-selected images



Security Test Results

54 firmware selected for manual investigation

Security Test Results

54 firmware selected for manual investigation

4 contains valid exploitation primitives

Security Test Results

Additional research must be done!