CFI-ANALYZER

Control Flow Hijacking detection through Intel PT

Diego Provinciani
diegoprovinciani@gmail.com

DProvinciani
DProvinciani

About Me…

McAfee - ASDC

- C/C++ Software Developer.
- OffSec TEAM leader.

Intel Security - ASDC

- C/C++ Software Developer.
- Script development – Bash, PowerShell and Python.

Universidad Nacional de Córdoba

- Computer Engineering.
- SysAdmin at Laboratorio de Computación
- ElementaryOS contributor.

Agenda

1. Return Oriented Programming (ROP).
2. Intel Processor Trace.
3. PT-DETECTOR detection mechanism.
4. Test Tool for validation.
5. Control Flow Integrity.
6. CFI-ANALYZER.

Return Oriented Programming (ROP)

Control-Flow Attack

Control-flow attacks allow an attacker to subvert the intended execution flow of a program by exploiting a vulnerability.

- **Code-Injection Attack**: Redirects the intended execution flow of a program to a previously injected malicious executable code.
- **Code-Reuse Attack**: Redirects the intended execution flow of a program to an unintended execution path inside the original program code.

NX Bit – W^X Mitigation

NO EXECUTABLE CODE ON THE STACK!!!

These features have been part of the Linux kernel mainline since the version 2.6.8.

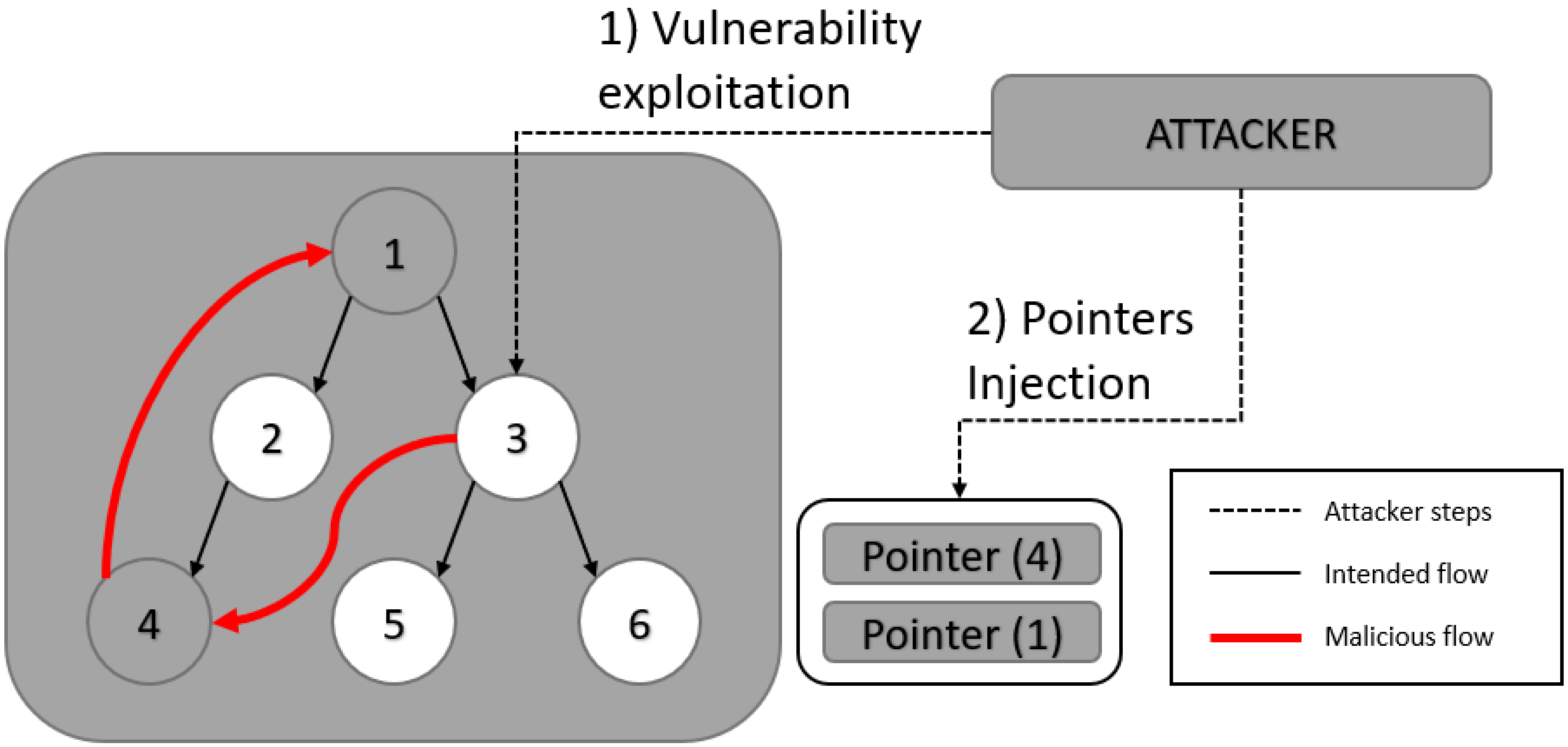
PaX NX technology can emulate NX functionality or use a hardware NX bit. The Linux kernel still does not ship with PaX.

From Mac OS X 10.4.4 (the first Intel release) onwards.

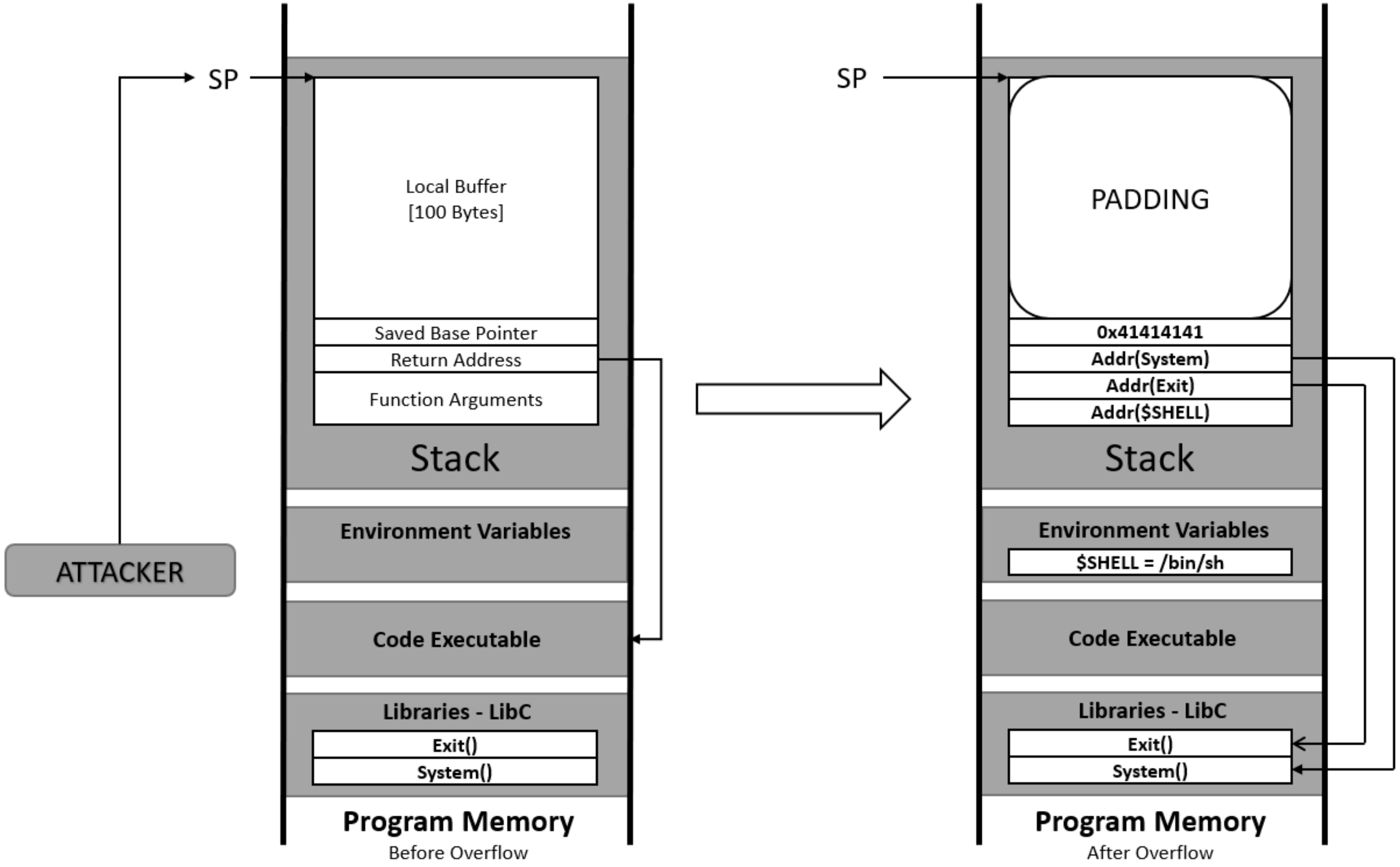
Starting on Windows XP Service Pack 2 (2004) and Windows Server 2003 Service Pack 1 (2005). "Data Execution Prevention" (DEP).

- Unfortunately, DEP/NX bit/W^X, whatever you want to call it, is easily subverted on its own.
- Turns out the guy that originally programmed the No Execute Stack patch also quickly developed a technique to defeat it.

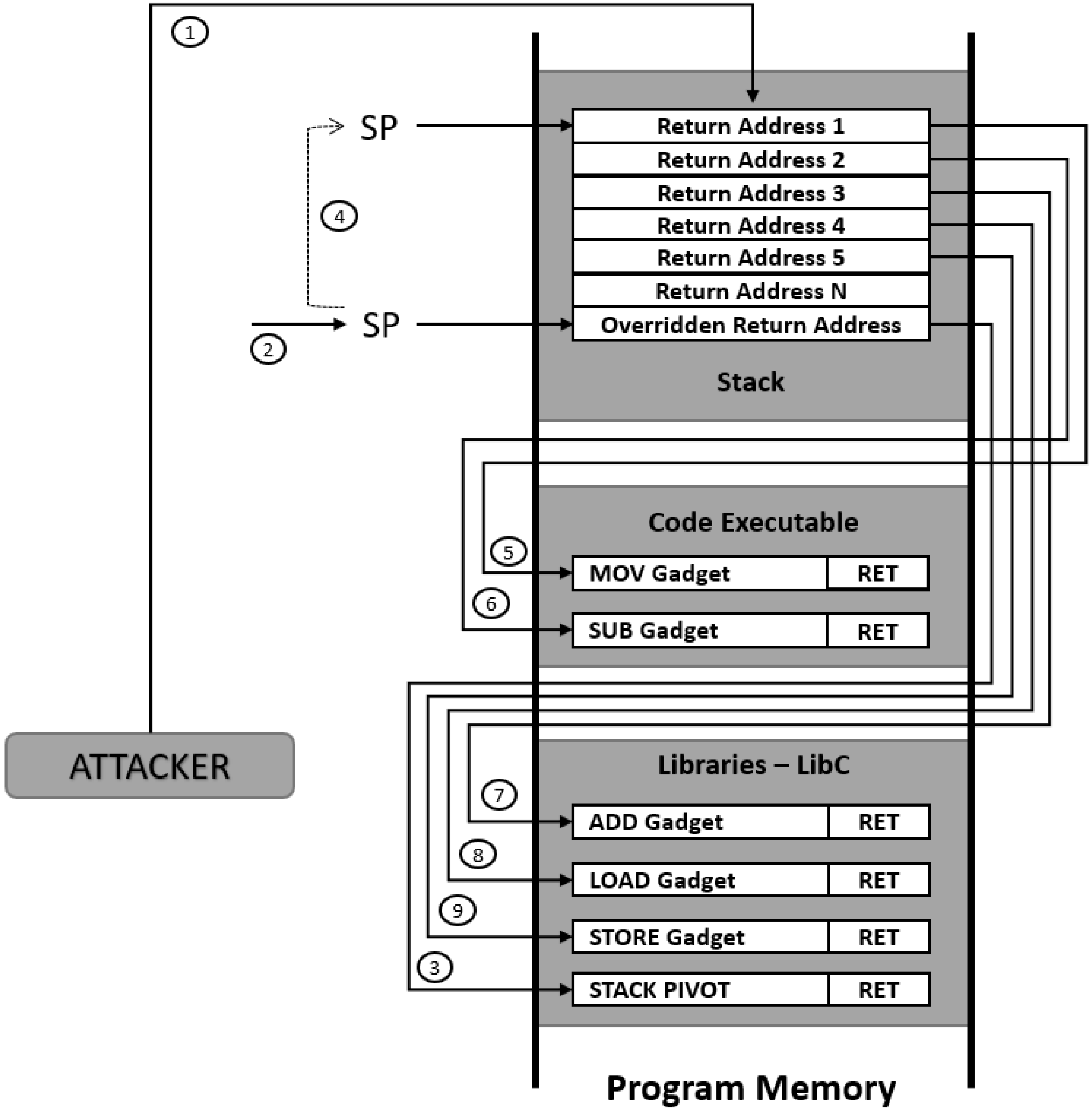
Code-Reuse Attack (CFG)



Ret2Libc… almost ROP



Return Oriented Programming (ROP)



How many gadgets are needed?

“Some researches shows that existing real-world ROP attacks have at least 17 gadgets, and the length of longest gadget chain of normal execution flows is 10. The threshold for the gadget chain length can be a number from 11 to 16 to reduce the false positive and false negative.”

Source: **“ROPecker: A Generic and Practical Approach For Defending Against ROP Attack”**

GOAL 1: Vulnapp detection

```c
//reads file into buffer and prints it
void printfile(char *filename) {
    char buf[64];
    FILE *fp;
    int filesize;

    fp = fopen(filename,"r");
    if(!fp) {
        printf("Error opening %s\n", filename);
        return;
    }

    fseek(fp, 0, SEEK_END);
    filesize = ftell(fp);
    fseek(fp, 0, SEEK_SET);

    //buffer overflow happens here
    fgets(buf, filesize+1, fp);

    fclose(fp);

    puts(buf);
}
```

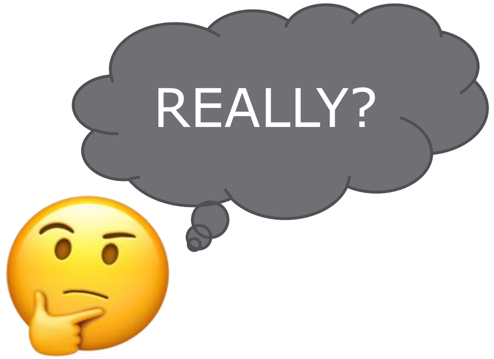
Let’s show
a demo!!!

https://github.com/ivanfratric/ropguard

Intel Processor Trace

Intel PT

- **Provides information about the software execution.** All the trace information to a memory buffer.
- **Makes use of special hardware** (minimum performance impact - officially 5%).



- **Filtering capabilities.**
  - Memory segments. (used for modules)
  - CR3. (filtering by process)

Intel PT

- **Information provided in “data packets”:**
  - Timing and synchronization packets.
  - Execution mode packets.
  - Flow execution packets.
- **Available from 5th Intel Processors Generation.**
- **Well documented on:**
  Intel® 64 and IA-32 Architectures Software Developer’s Manual - Volume 3 - Chapter 35.

Intel PT - Applications

- **Process debugging and performance analysis.**
  Capabilities included in Intel® System Studio.

- **Vulnerability Discovery.**
  Richard Johnson & Andrea Alevi: Harnessing Intel Processor Trace on Windows for Vulnerability Discovery.

- **Other security applications.**
  Shlomi Oberman and Ron Shina: COFI Break.
  Microsoft Research: GRIFFIN.
  Alex Ionescu and Windows 10 RS5: WinIPT.
  Shanghai University Research: FlowGuard.

Intel PT – Packet Summary

- **PSB** → ‘Heartbeats’ generated at regular intervals.
- **TNT** → Conditional Branch Taken or Not Taken.
- **TIP** → Target for indirect branches, interrupts etc.
- **FUP** → Source for asynchronous events and unintended execution flows.
- **PIP** → Paging Information Packet. CR3 modifications.
- **MODE** → Current CPU mode (16/32/64 bits).
- And more…

Let’s show a demo!!!

Intel PT – TNT Packet

6 Conditional branches

	7	6	5	4	3	2	1	0	
0	1	B_1	B_2	B_3	B_4	B_5	B_6	0	Short TNT

47 Conditional branches

	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	1	0	Long TNT
1	1	0	1	0	0	0	1	1	
2	B_{40}	B_{41}	B_{42}	B_{43}	B_{44}	B_{45}	B_{46}	B_{47}	
3	B_{32}	B_{33}	B_{34}	B_{35}	B_{36}	B_{37}	B_{38}	B_{39}	
4	B_{24}	B_{25}	B_{26}	B_{27}	B_{28}	B_{29}	B_{30}	B_{31}	
5	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}	
6	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	
7	1	B_1	B_2	B_3	B_4	B_5	B_6	B_7	

Intel PT – TIP Packet

Also valid for: TIP.PGE and TIP.PGD.

	7	6	5	4	3	2	1	0
0	IPBytes			0	1	1	0	1
1	TargetIP[7:0]							
2	TargetIP[15:8]							
3	TargetIP[23:16]							
4	TargetIP[31:24]							
5	TargetIP[39:32]							
6	TargetIP[47:40]							
7	TargetIP[55:48]							
8	TargetIP[63:56]							

Intel PT – FUP Packet

	7	6	5	4	3	2	1	0
0	IPBytes			1	1	1	0	1
1	IP[7:0]							
2	IP[15:8]							
3	IP[23:16]							
4	IP[31:24]							
5	IP[39:32]							
6	IP[47:40]							
7	IP[55:48]							
8	IP[63:56]							

Intel PT – Trace Example 1

```
int sum (int a, int b, int c) {
    int result = 0;

    result = a + b + c;
    return result;

}


int main (int argc, char * argv []) {

    int a = 1;
    int b = 2;
    int c = 3;
    int result = 0;

    result = sum (a, b, c);
    printf ("%d + %d + %d = %d", a, b, c, result);

    return 0;

}
```

Intel PT – Trace Example 1

```
tip.pge   2: ????????004012bd --> return from __get_initial_narrow_environment
tnt.8     !!
tip       1: ????????004010a3 --> return from suma
tip.pgd   2: ????????76decf70 --> call to acrt_iob_func
tip.pge   2: ????????004010f3 --> return from acrt_iob_func
tnt.8     !
tip.pgd   2: ????????76dd37c0 --> call to __stdio_common_vprintf
mode.exec cs.d
tip.pge   3: 0000000000401065 --> return from __stdio_common_vprintf
tip       1: ????????????10fc --> return from _vprintf_l
tip       1: ????????????10c0 --> return from printf
tip       1: ????????????12c7 --> return from main
```

Intel PT – Trace Example 2

```
typedef int (*foo) (int, int, int);

int sum (int a, int b, int c) {
    int result = 0;

    result = a + b + c;
    return result;
}


int main (int argc, char * argv []) {

    int a = 1;
    int b = 2;
    int c = 3;
    int result = 0;
    foo calc = sum;

    result = calc (a, b, c);
    Printf ("%d + %d + %d = %d", a, b, c, result);

    return 0;
}
```

Intel PT – Trace Example 2

```
tip.pge   2: ????????004012bd --> return from __get_initial_narrow_environment
tip       1: ????????00401010 --> call to calc pointer
tnt.8     !!
tip       1: ????????004010a3 --> return from calc pointer
tip.pgd   2: ????????76decf70 --> call to acrt_iob_func
tip.pge   2: ????????004010f3 --> return from acrt_iob_func
tnt.8     !
tip.pgd   2: ????????76dd37c0 --> call to __stdio_common_vprintf
mode.exec cs.d
tip.pge   3: 0000000000401065 --> return from __stdio_common_vprintf
tip       1: ????????????10fc --> return from _vprintf_l
tip       1: ????????????10c0 --> return from printf
tip       1: ????????????12c7 --> return from main
```

Intel PT – Trace Example 3

```
typedef int (*foo) (int, int, int);

int sum (int a, int b, int c) {
    int result = 0;
    result = a + b + c;

    __asm {mov dword ptr ss : [ebp + 4], 0x41414141}
    return result;
}


int main (int argc, char * argv []) {

    int a = 1;
    int b = 2;
    int c = 3;
    int result = 0;
    foo calc = sum;

    result = calc (a, b, c);
    Printf ("%d + %d + %d = %d", a, b, c, result);

    return 0;
}
```

Intel PT – Trace Example 3

```
tip.pge   2: ????????004012bd --> return from __get_initial_narrow_environment
tip       1: ????????00401010 --> call to calc pointer
tnt.8     !!
tip       1: ????????41414141 --> return from calc pointer
fup       1: ????????41414141 --> return from calc pointer exception
```

WHY TIP AND FUP???

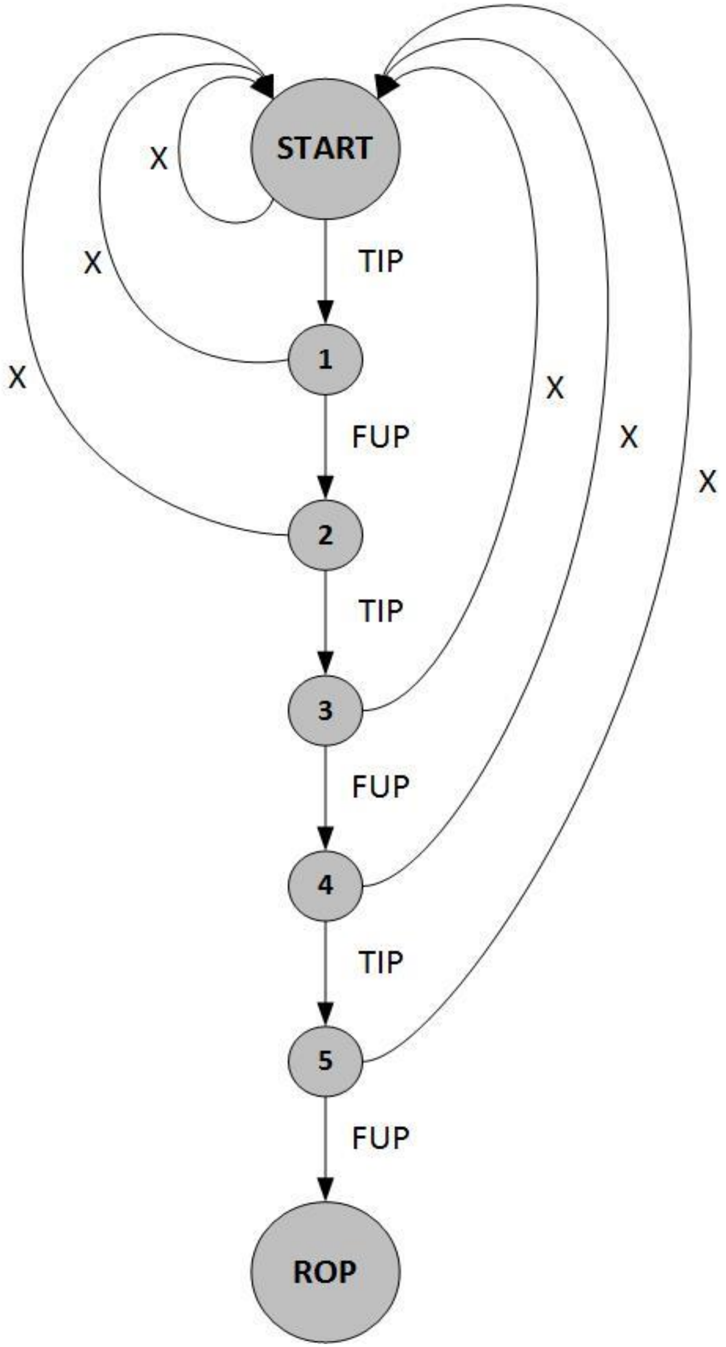
REMEMBER!!!

tip → shows the “TARGET” instruction pointer
fup → shows the exception “STARTING ADDRESS”.

PT-DETECTOR

PT-DETECTOR detection mechanism

A sequence of three couple of packets TIP + FUP is enough to confirm the presence of a ROP chain being executed.
Just two triggers some false positives.



Let’s show a demo!!!

PT-DETECTOR Limitations

The points below mostly refers to limitations of the technology itself

- It can currently monitor any thread of a single user space process.
- Kernel code monitoring is possible but not yet implemented on PT-DETECTOR.
- It increases 10% the usage of the running CPU core when monitoring a single process.
- We don’t have visibility of the instructions being executed.

Test Tool for Validation

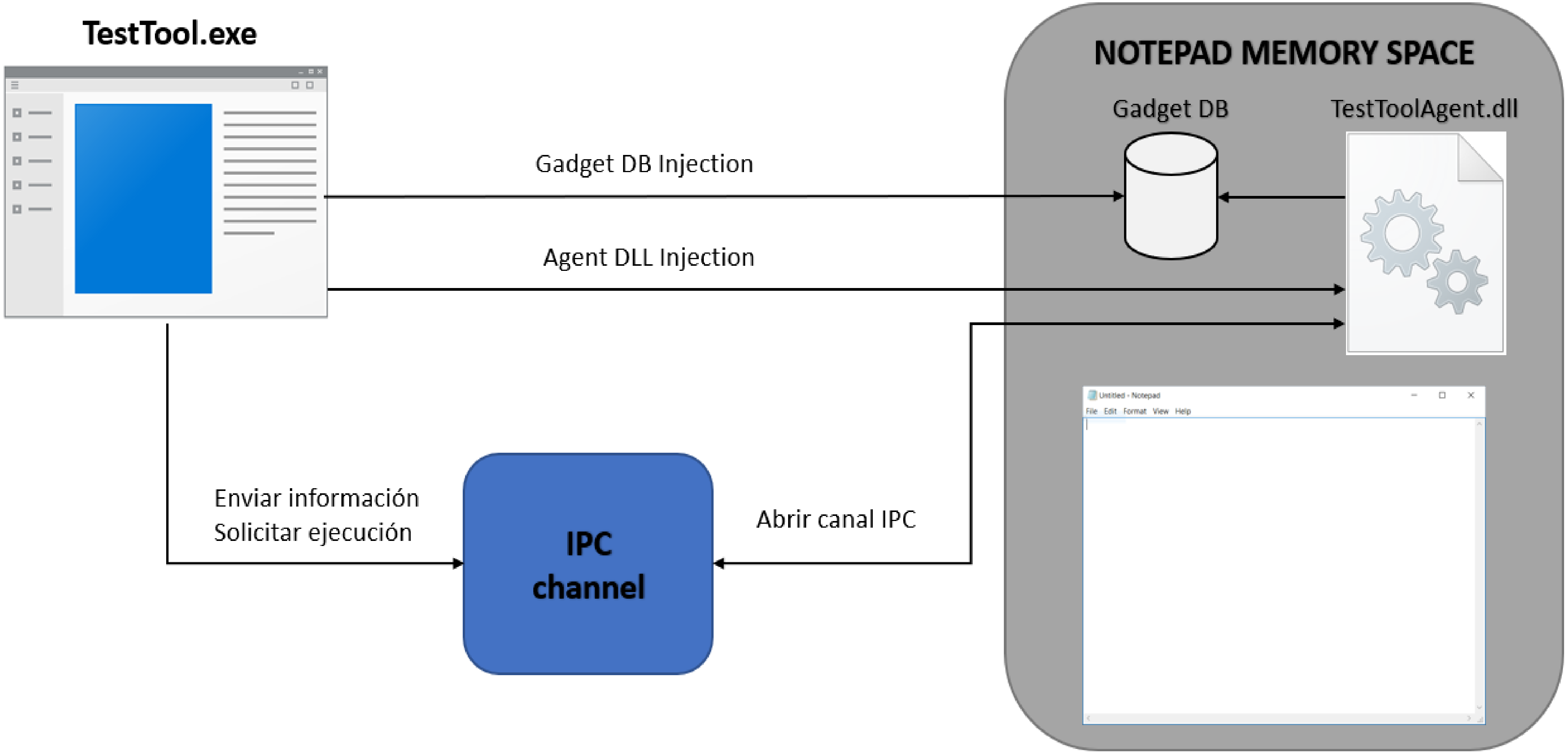
Why this Test Tool?

- Exploits are very dependent on the environment.

- Limited number of targets to attack.

- Is not user friendly to show Hex values in a demo.

Test Tool high level architecture



Test Tool – DLL Injection

```
HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION | PROCESS_VM_WRITE,
FALSE, targetPID);

LPVOID pszLibFileRemote = (PWSTR)VirtualAllocEx(hProcess, NULL,
dllPathNameSize, MEM_COMMIT, PAGE_READWRITE);

WriteProcessMemory(hProcess, pszLibFileRemote,
(PVOID)codeToInject.c_str(), dllPathNameSize, NULL);

PTHREAD_START_ROUTINE pfnThreadRtn =
(PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("kernel32
.dll")), "LoadLibraryW");

HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0, pfnThreadRtn,
pszLibFileRemote, 0, NULL);
```

Test Tool execution summary

1. Get target PID and test case file.
2. Inject TestToolAgent.dll into the target memory space.
3. Validate test case file.
4. Assembly the ROP gadgets using Keystone.
5. Request executable memory into the target process
6. Write the assembled gadgets on this piece of memory
7. Build a payload with the address of each gadget
8. Send the payload to the target to be executed as a buffer overflow.

Let’s show a demo!!!

Control Flow Integrity (CFI)

Control Flow Integrity

- Mitigation technique that ensure the right execution flow of a program.

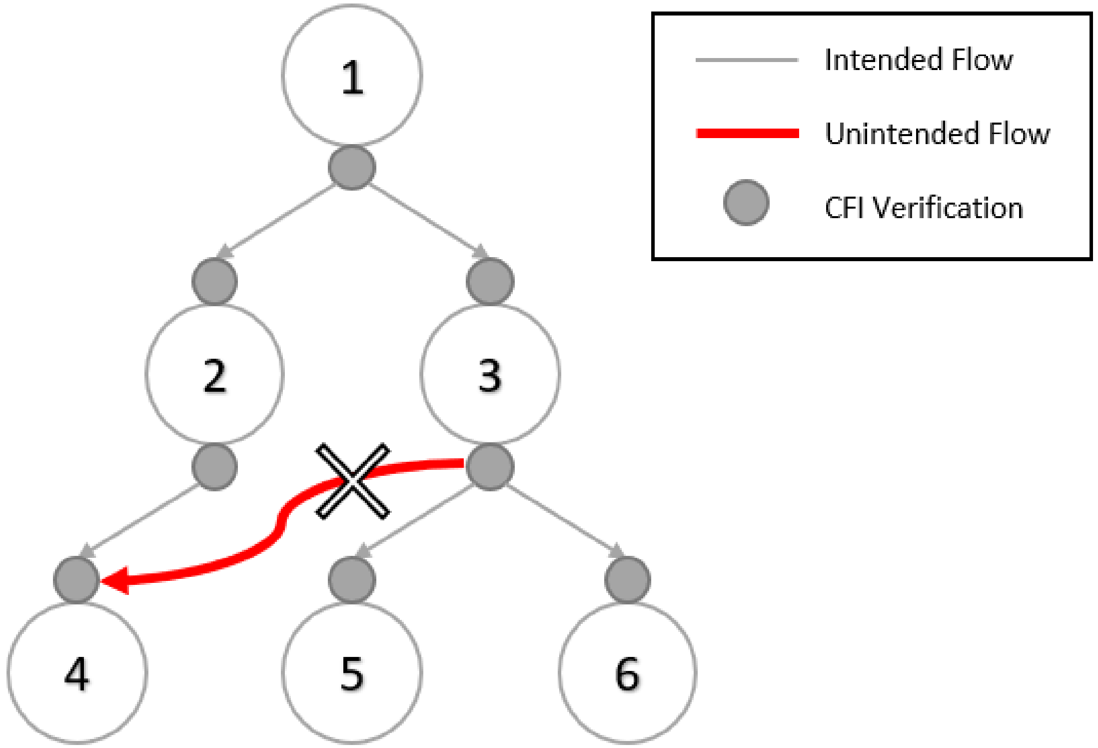
- Based on the analysis of Control Flow Graphs.

- The flow execution must follow a right path in a Control Flow Graph.

Control Flow Integrity

TAG BASED IMPLEMENTATION:

1. Enabled by compiler.
2. Binary instrumentation.

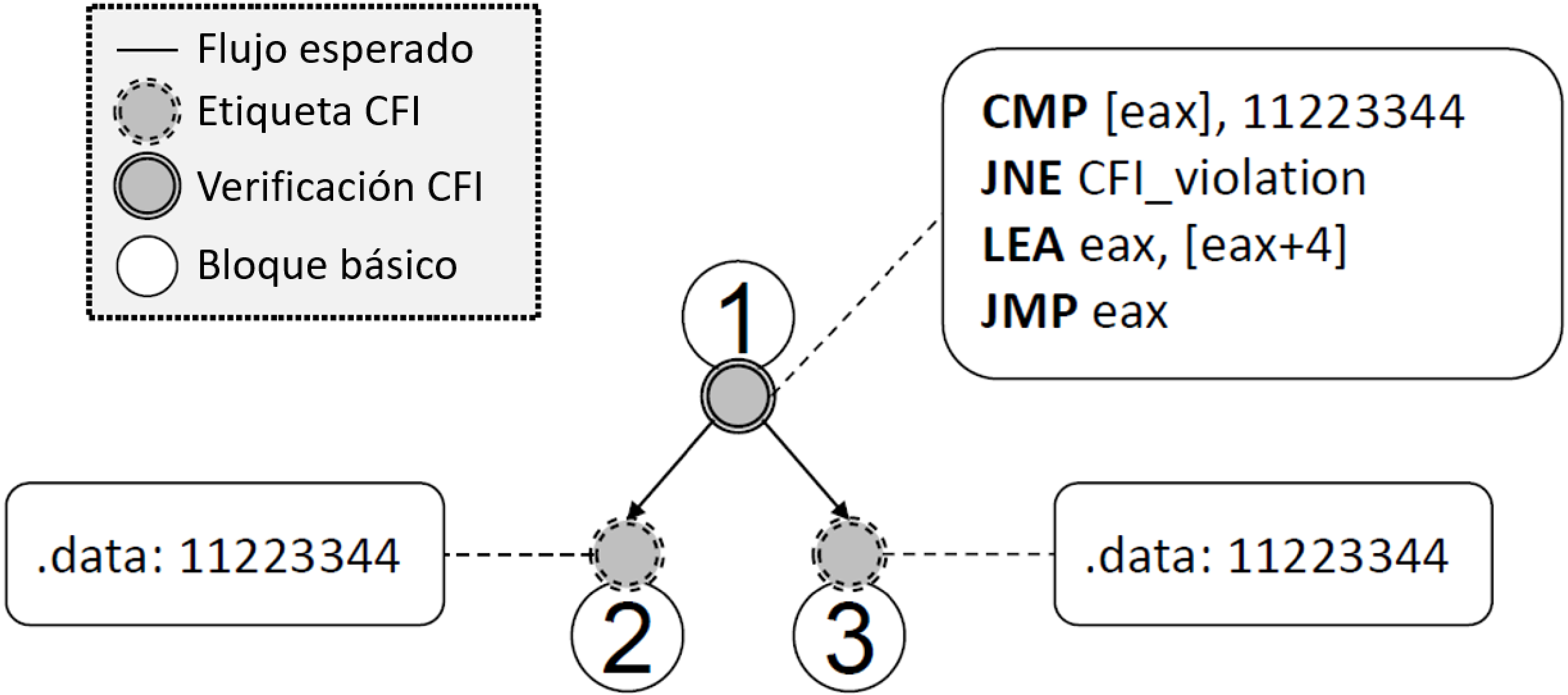


Control Flow Integrity

There are three possibilities to apply CFI:

1. CFI for indirect JUMP instructions
2. CFI for indirect CALLs
3. CFI for function returns

Control Flow Integrity – JUMPs and CALLs



Microsoft CFG

- CFI implementation just for Indirect CALLs.
- Target address is passed to the **_guard_check_icall function**.
- In Windows 10, which does have CFG support, it points to **ntdll!LdrpValidateUserCallTarget**. This function takes a target address as argument and does the following:
  - Access a bitmap (called **CFGBitmap**) which represents the starting location of all the functions in the process space.
  - Then, convert the target address to one bit in **CFGBitmap**.

Microsoft CFG Implementation

```
typedef int(*fun_t) (int);

int foo(int a)
{
    printf("Hello team member: %d\n", a);
    return a;
}

class CTargetObject
{
public:
    fun_t _fun;
};

int main()
{
    int i = 0;
    CTargetObject* o_array = new CTargetObject[5];

    for (i = 0; i < 5; i++)
        o_array[i]._fun = foo;

    o_array[0]._fun(1);

    return 0;
}
```

Building with **/guard:cf**

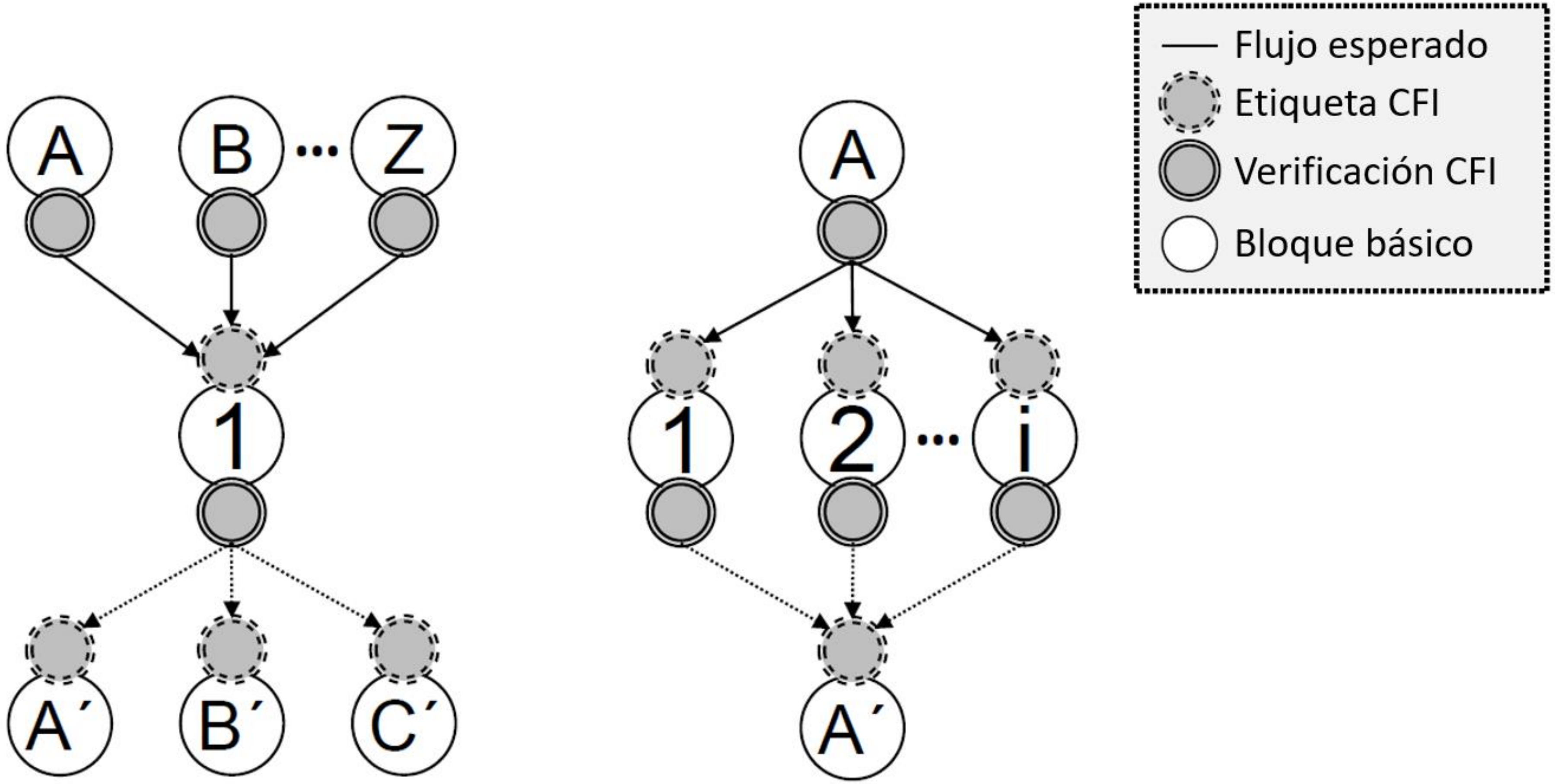
```
mov esi,esp
push 1
mov ecx,4
imul edx,ecx,0
mov eax,dword ptr ss:[ebp-8]
mov ecx,dword ptr ds:[eax+edx]
mov dword ptr ss:[ebp-10],ecx
mov edi,esp
mov ecx,dword ptr ss:[ebp-10]
call dword ptr ds:[<_guard_check_icall_fptr>]
cmp edi,esp
call <cfg.ILT+688(__RTC_CheckEsp)>
call dword ptr ss:[ebp-10]
add esp,4
cmp esi,esp
call <cfg.ILT+688(__RTC_CheckEsp)>
xor eax,eax
pop edi
pop esi
add esp,10
cmp ebp,esp
call <cfg.ILT+688(__RTC_CheckEsp)>
mov esp,ebp
pop ebp
ret
```

Microsoft CFG Limitations

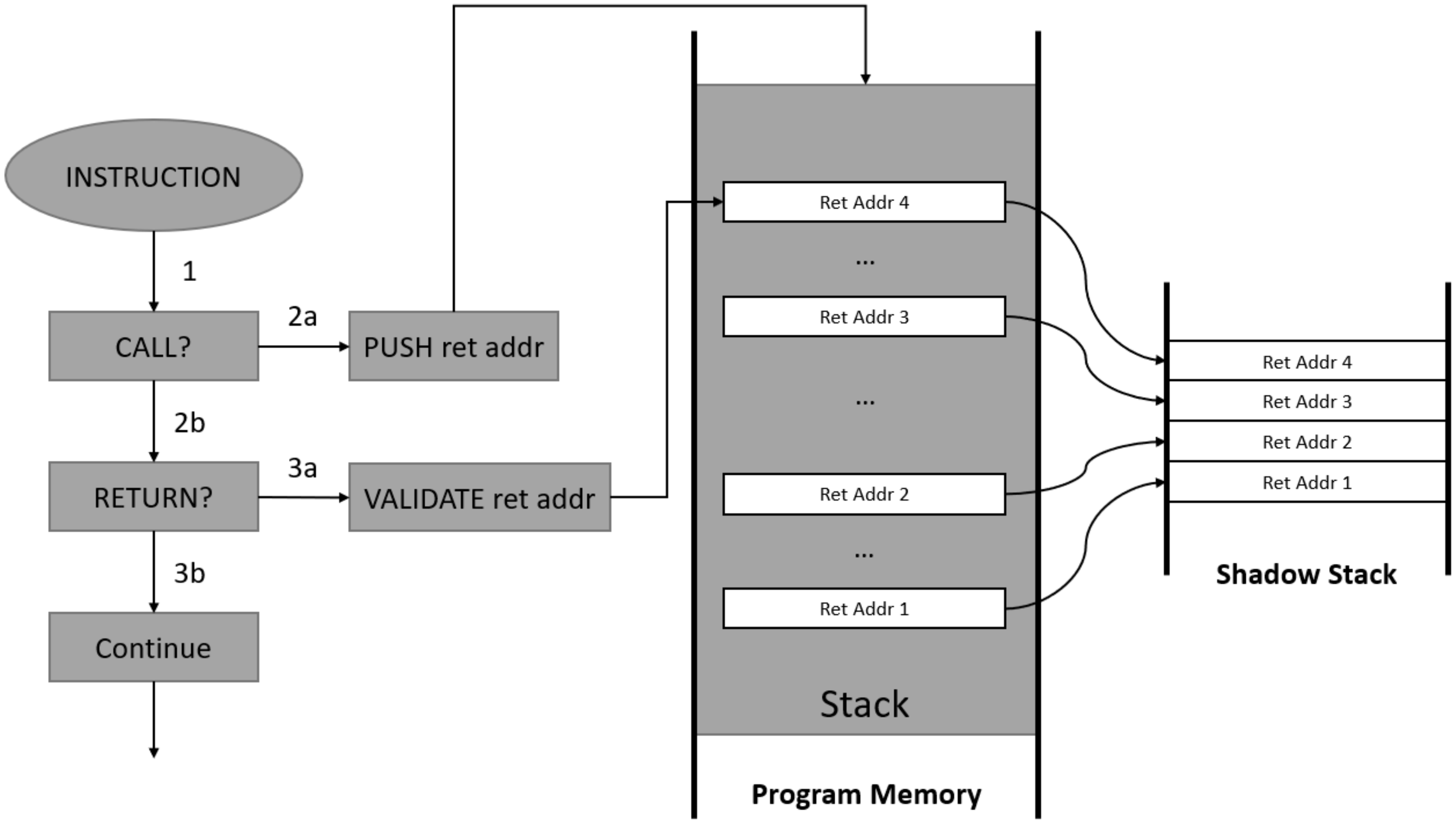
Mitigation	In scope	Out of scope
Control Flow Guard (CFG)	Techniques that make it possible to gain control of the instruction pointer through an indirect call in a process that has enabled CFG.	• Hijacking control flow via return address corruption • Bypasses related to limitations of coarse-grained CFI (e.g. calling functions out of context) • Leveraging non-CFG images • Bypasses that rely on modifying or corrupting read-only memory • Bypasses that rely on CONTEXT record corruption • Bypasses that rely on race conditions or exception handling • Bypasses that rely on thread suspension • Instances of missing CFG instrumentation prior to an indirect call • Code replacement attacks

https://www.microsoft.com/en-us/msrc/bounty-mitigation-bypass?rtc=1

Control Flow Integrity – RETs problem



Control Flow Integrity – RETs shadow stack

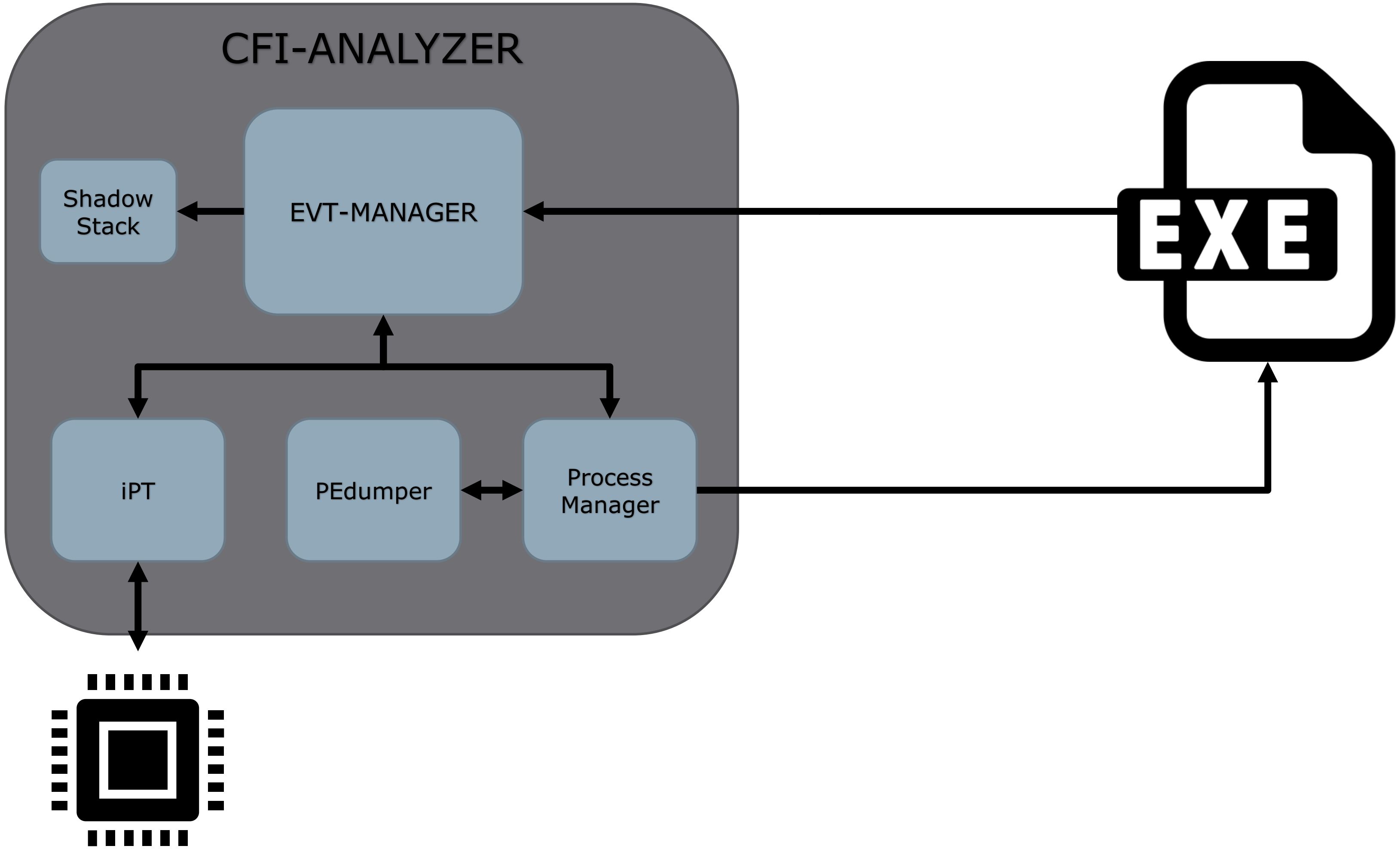


Intel CET

- Control-flow Enforcement Technology (CET).
- Two components:
  - **Shadow Stack (SHSTK)**: The RET instruction pops the return address from both stacks (the shadow and the owned by the process) and compares them. If the return addresses from the two stacks do not match, the processor signals a control protection exception (#CP).
  - **Indirect Branch Tracking (IBT)**: The CPU implements a state machine that tracks indirect jmp and call instructions. When one of these instructions is seen, the state machine moves from IDLE to WAIT_FOR_ENDBRANCH state. In WAIT_FOR_ENDBRANCH state the next instruction in the program stream must be an ENDBRANCH. If an ENDBRANCH is not seen the processor causes a control protection exception (#CP), else the state machine moves back to IDLE state.

CFI-ANALYZER

CFI-ANALYZER high level architecture



CFI-ANALYZER execution summary

1. Process Manager starts the process.
2. Process Manager sets a BP in the entry point.
3. Event Manager receives de entry point signal and orders the process manager to set the remaining breakpoints in all CALL and RET instructions. Then it starts the trace and continues.
4. After hitting each breakpoint Event Manager stops the trace, analyzes the data, enables the trace, and continue.

Let’s show a demo!!!

CFI-ANALYZER – Next Steps

- JUMPs and CALLs CFI support.
- Multiple thread support.
- Kernel code support.
- Per-core and per-process support.

Any question?

THANKS, EVERYBODY!!!

Diego Provinciani
diegoprovinciani@gmail.com

DProvinciani
DProvinciani

BSIDES CÓRDOBA 2019