Augmenting Binary Analysis with Python and Pin

January 14th, 2014

@ancat, @1blankwall1

Who are we?

About Us

- Omar
  - Recent graduate of NYU
  - Security engineer at Etsy
- Tyler
  - Studies at NYU
  - Security researcher at SilverSky

Etsy

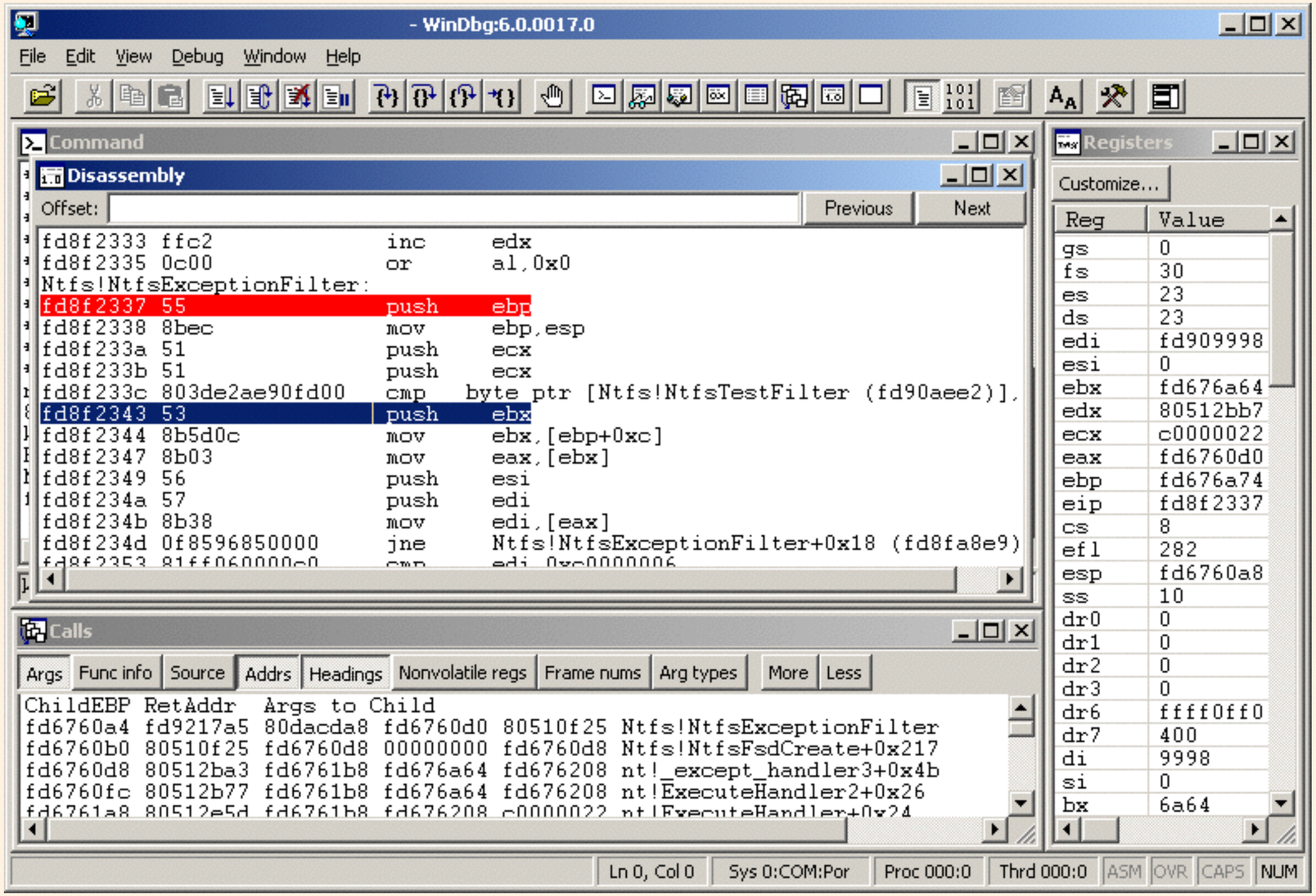
SilverSky
SECURITY FROM THE CLOUD

What is binary analysis?

What is binary analysis?

- *Binary:* A file containing all the resources and native code needed for a program to execute
- *Analysis:* To make sense of an application when the original intentions are not clear or known

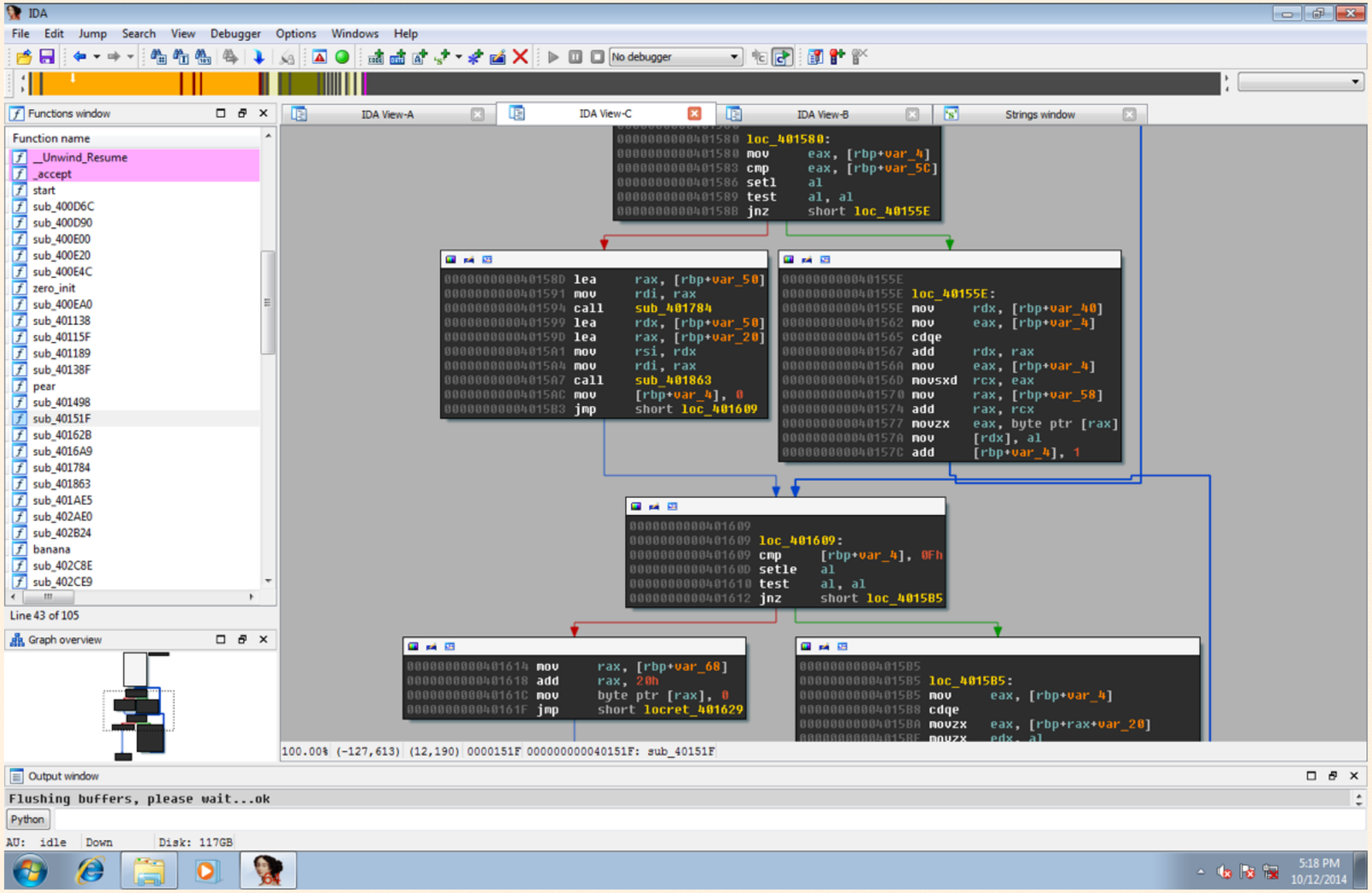
Using a debugger (WinDbg, GDB, Immunity, etc)



Simply observing the execution of a binary

```
$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
qwertyuiop

BOOM!!!
The bomb has blown up.
$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
```

Reading disassembly output (IDA, objdump, etc)



Running /usr/bin/strings on a binary

```
$ strings ./elysium
/lib/ld-linux.so.2
libcrypto.so.1.0.0
EVP_DecryptFinal_ex
EVP_aes_128_cbc
EVP_DecryptInit_ex
RAND_pseudo_bytes
EVP_EncryptFinal_ex
EVP_CIPHER_CTX_init
EVP_DecryptUpdate
EVP_EncryptInit_ex
SHA1
EVP_EncryptUpdate
libc.so.6
_IO_stdin_used
setuid
socket
strcpy
exit
htons
[-] Send Fail
1) Get informations <name>
2) List units
3) Add medical units <count>
4) Add military units <count>
5) Add social units <count>
```

Static Analysis

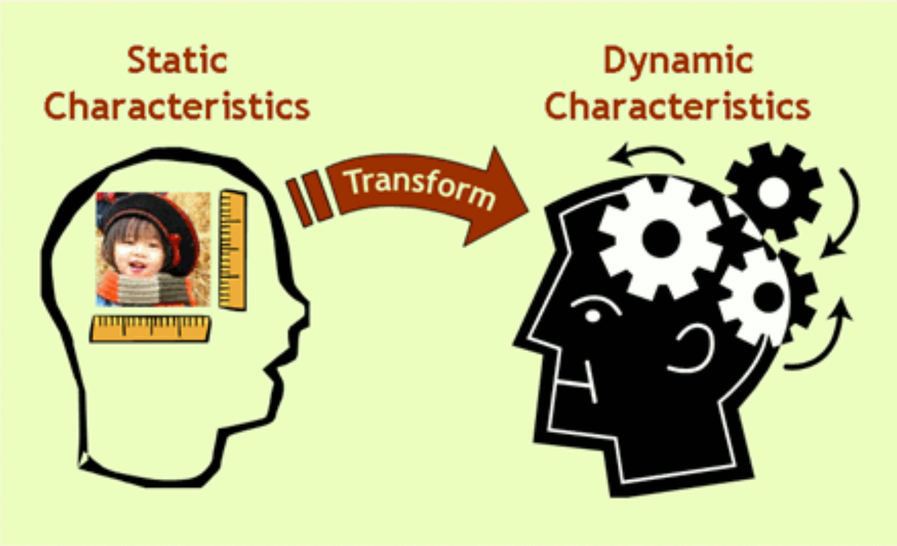
- Reading disassembly output (IDA, objdump, etc)
- Running /usr/bin/strings on a binary

Dynamic Analysis

- Using a debugger (WinDbg, gdb, Immunity, etc)
- Simply observing the execution of a binary

Static vs Dynamic

- Speed
- Level of Understanding
- Code Coverage
  - Static can cover 100% of the code (good or bad?)
  - Dynamic can be accurate due to run time information



Introducing...

Dynamic Binary Instrumentation

Dynamic Binary Instrumentation

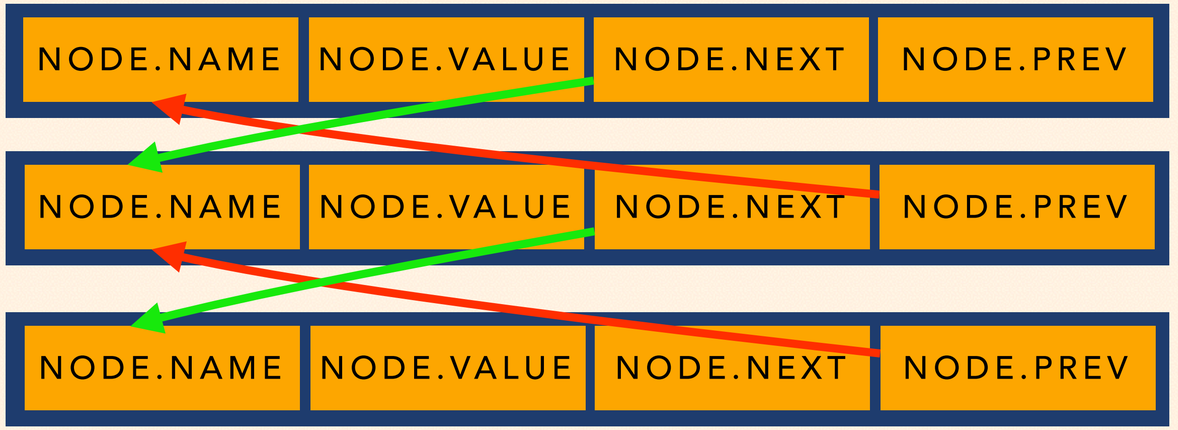
- A technique to modify the behavior of programs based on certain conditions during execution
  - Sometimes done by modifying the code before starting the program
  - For example, an INT3 instruction on x86 used by debuggers, or less specifically, trampolines

Debugger Scripting

- GDB & LLDB
  - Scriptable using Python - Unix only (mostly)
- WinDBG
  - Scriptable using Python (somewhat) - Windows only
- VDB
  - Entirely Python API - Windows and and Unix support

Debugger Scripting

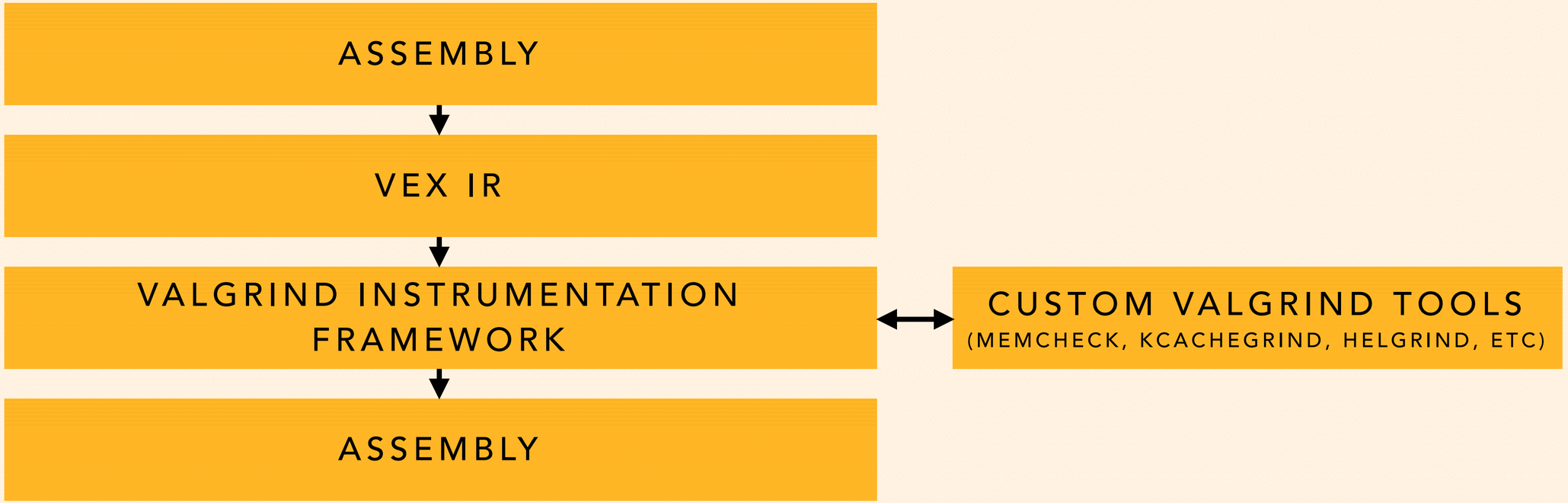
```
define structs
    set $target = $root
    set $limit = 0
    while $target
        printf "[0x%x] node.name=0x%x; node.value=0x%x; node.next=0x%x; node.prev=0x%x\n",
            $target, *($target), *($target+4), *($target+8), *($target+0xc)
        set $old_target = $target
        set $target = *($target+8)

        if $old_target == $target
            set $limit = $limit + 1
        end

        if $limit > 10
            printf "Infinite loop?\n"
            set $target = 0
        end
    end
end
```



DBI Frameworks

- Valgrind
  - GPL'd system for debugging and profiling Linux programs
  - Automatically detects many memory management and threading bugs
  - Works on x86/Linux, AMD64/Linux and PPC32/Linux
  - Focused on Safe and Reliable Code
    - Developer tool used for finding code errors

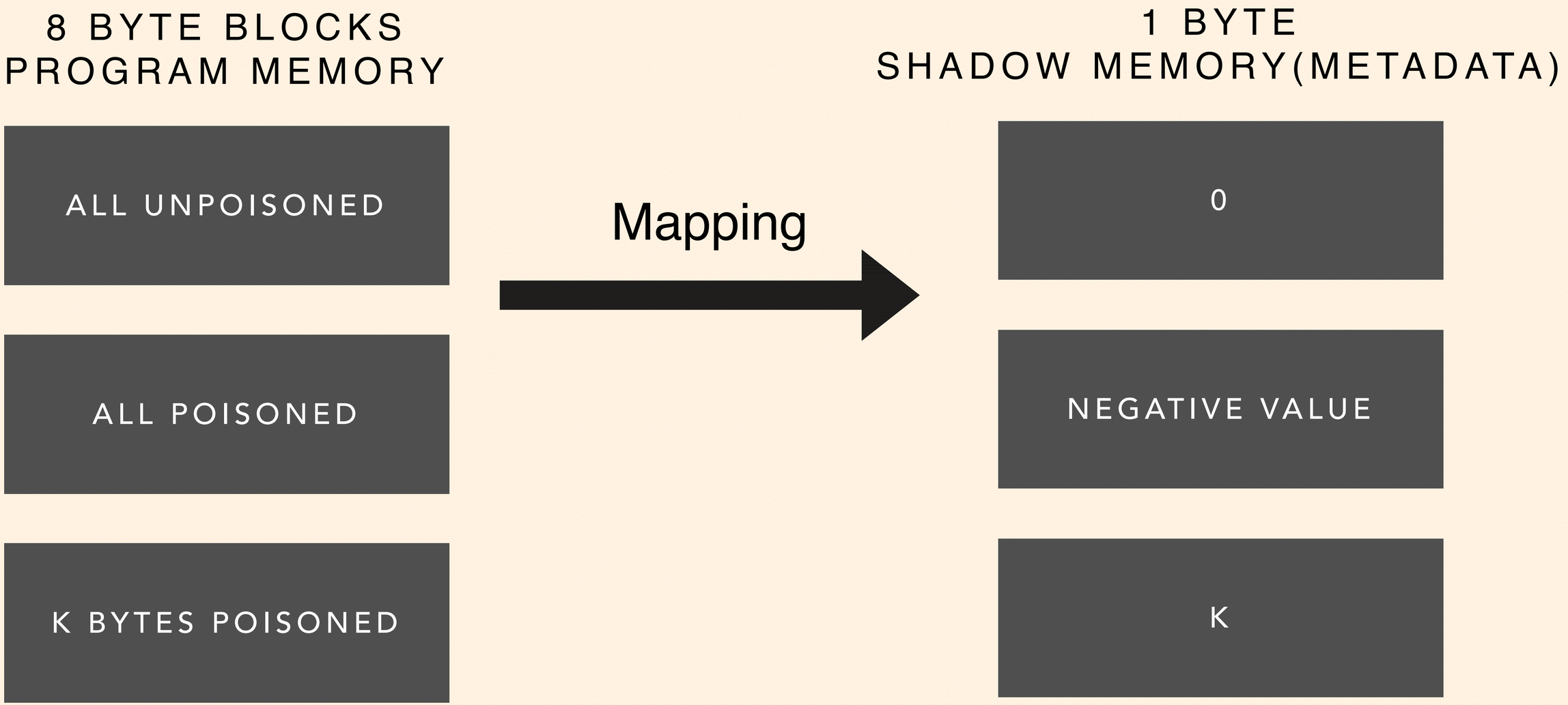
DBI Frameworks



DBI Frameworks

- Address Sanitizer
  - Fast memory error detector
  - The tool consists of a compiler instrumentation module (currently, an LLVM pass) and a run-time library which replaces the malloc function
  - Works on x86 Linux, and Mac, and ARM Android
  - Focused on bugs
    - Heap/Stack Buffer overflows and Use After Free

Address Sanitizer Algorithm



DBI Frameworks

```
=================================================================
==5472==ERROR: AddressSanitizer: heap-use-after-free on address 0x60300000eff8 at pc 0x41ef66 bp 0x7fffa5849fb0 sp 0x7fffa5849f88
READ of size 4 at 0x60300000eff8 thread T0
    #0 0x41ef65 (/tmp/a.out+0x41ef65)
    #1 0x7f2c68b10658 (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.18+0x99658)
    #2 0x42f6d8 (/tmp/a.out+0x42f6d8)
    #3 0x7f2c681d7a54 (/lib/x86_64-linux-gnu/libc-2.17.so+0x21a54)
    #4 0x42f2dc (/tmp/a.out+0x42f2dc)
0x60300000eff8 is located 24 bytes inside of 28-byte region [0x60300000efe0,0x60300000effc)
freed by thread T0 here:
    #0 0x421654 (/tmp/a.out+0x421654)
    #1 0x7f2c68b3551e (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.18+0xbe51e)
    #2 0x7f2c681d7a54 (/lib/x86_64-linux-gnu/libc-2.17.so+0x21a54)
previously allocated by thread T0 here:
    #0 0x421494 (/tmp/a.out+0x421494)
    #1 0x7f2c68b353c8 (/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.18+0xbe3c8)
    #2 0x5
Shadow bytes around the buggy address:
  0x0c067fff9da0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff9db0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff9dc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff9dd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff9de0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c067fff9df0: fa fa fa fa fa fa fa fa fa fa fa fa fd fd fd[fd]
  0x0c067fff9e00:fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff9e10: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff9e20: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff9e30: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c067fff9e40: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:     fa
  Heap right redzone:    fb
  Freed heap region:     fd
  Stack left redzone:    f1
  Stack mid redzone:     f2
  Stack right redzone:   f3
  Stack partial redzone: f4
  Stack after return:    f5
  Stack use after scope: f8
  Global redzone:        f9
  Global init order:     f6
  Poisoned by user:      f7
  ASan internal:         fe
==5472==ABORTING
```

DBI Frameworks

- DynamoRIO
  - Runtime code manipulation system that supports code transformations on any part of a program at runtime
  - Works on x86/AMD64 Linux Mac, and Windows
  - Transparent, and comprehensive manipulation of unmodified applications running on stock operating systems
  - Direct Competitor to Pin :-!

What *is* Pin?

- Pin allows user to insert arbitrary code into an executable right after it is loaded into memory
- Generates code from a “PinTool” used to “hook” instructions and calls
- Pin is the framework
- PinTools are the interface
  - The mechanism that decides where and what code is inserted
  - The code to execute at insertion points

Why Pin?

Intel's Pin

- Amazing documentation
- Same exact API works for Windows and Unix
- Extremely popular
- Nothing needs to be recompiled to be used with Pin

It's easy to get started

- Large repo of well commented sample tools come with Pin
- Documentation is generally easy to follow
- Installation is a piece of cake

It can be as granular as you need it to be

- Simple hook/callback system
  - function calls
  - basic blocks
  - instructions
  - and so on

Mostly personal preference, though

Why not Pin?

- The Pin API uses C++
  - Not a huge deal, but can be inconvenient during a time crunch (ctf)
  - Harder to prototype
- Slower than other DBI Frameworks
- Not as granular as other solutions
  - Harder to do more advanced binary analysis techniques such as taint tracing

Awesome but what can Pin do?

Popular Uses

- The Pin API has been used extensively in industry
- Most notably Microsoft Blue Hat (2012) Winner kBouncer (Vasilis Pappas)
  - Efficient and fully transparent ROP mitigation technique
  - Very similar to second place ROPGuard (Ivan Fratric)
    - Used in Microsofts EMET protection system
- IDA 6.4 and above includes a pin tool for tracing code in the debugger

Cool… WHERE ARE MY BUGS?!

- Pin can be used to find many different classes of bugs
- Most can be found by using the right kind of instrumentation
  - Format Strings
    - Analyze parameters passed to formatting functions
  - Buffer Overflows
    - Analyze memory read and write instructions
  - Misused Memory Allocation (Double Frees or UAF)
    - Analyze memory allocation functions (malloc/free) and memory writes

Misused Heap Allocations

- How to find these dynamically?
  - Keep track of all malloc calls and the addresses returned
  - Maintain state: Freed or In use and size
  - When a memory read or write happens, if the target is on the heap, verify that the memory is a valid place to be read from or written to

D-d-d-d-d-demo!

- Pin C++ Heap Overflow Demo

Pin

- Wow, Pin is really cool!
- But, wait! Pin is a mess!
  - Correction, C++ is a mess :P
- Lots of necessary boilerplate code
- Hard to prototype quickly
- Difficult to understand

C++

```
RTN mallocRtn = RTN_FindByName(img, MALLOC);
if (RTN_Valid(mallocRtn))
{
    RTN_Open(mallocRtn);

    // Instrument malloc() to print the input argument value and the return value.
    RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR) Arg1Before,
                   IARG_ADDRINT, MALLOC,
                   IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                   IARG_END);
    RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR) MallocAfter,
                   IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);

    RTN_Close(mallocRtn);
}
```

Python

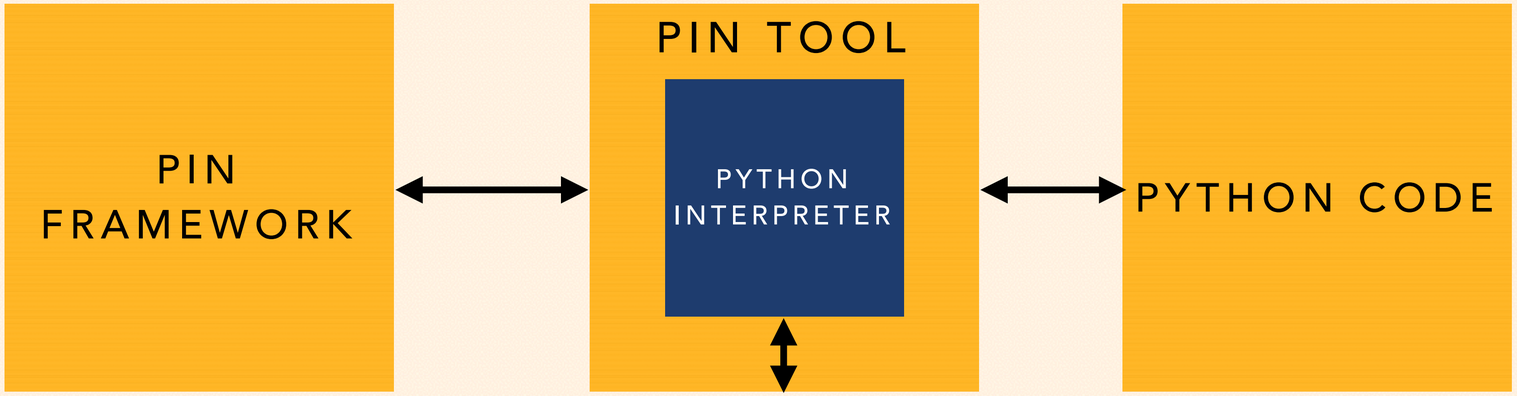
```
rtn = pin.RTN_FindByName(img, "malloc")
if pin.RTN_Valid(rtn):
        pin.RTN_Open(rtn)
        pin.RTN_InsertCall(pin.IPOINT_BEFORE, "malloc", rtn, 1, malloc_before)
        pin.RTN_InsertCall(pin.IPOINT_AFTER, "malloc", rtn, 1, malloc_after)
        pin.RTN_Close(rtn)
```

C++ vs Python

- Python
  - Simpler
  - Cleaner
  - No need for recompilation every time
  - Extensive libraries and support

Python-Pin

- Essentially, a python interpreter embedded within a PinTool
  - “Virtual” pin module exposed to the python script
  - Enables access to most of Pin’s functionality from within python
  - Quick and easy to write PinTools
  - Enables seamless integration with other Python modules
    - Z3py, PIL, SciPy, etc



Python-Pin Demo

- Use after free and heap overflow detection
- Transparent socket logging
- Basic utility demos

Basic Heap Overflow and UAF Protection

- USER CALLS MALLOC (CALLOC, REALLOC ETC…)
- PIN HOOKS ALLOCATION FUNCTIONS AND ADJUST REQUESTED SIZE TO ALLOW FOR CANARY ALLOCATIONS
- HOOKS RETURN VALUE AND ADJUSTS THE SIZE AS WELL AS SETTING ADDRESS'S WITH CANARY VALUE
- CHECKS HEAP READS AND WRITES TO ENSURE CANARY VALUE IS NOT PRESENT

Basic Heap Overflow and UAF Protection

FREE

- PIN HOOKS FREE FUNCTION
- ADDS EVERY FREED BLOCK TO THE FREE LIST
- VERIFIES HEAP ACCESS AGAINST THE FREE LIST BY HOOKING READS AND WRITES

Basic Heap Overflow and UAF Protection

LIMITATIONS:

- LARGE COMPUTATION TIME TO CHECK THE FREE LIST EVERY TIME
- CHICKEN OR THE EGG PROBLEM
  - PIN BEGINS HOOKING FREES AND ALLOCATIONS AT A VARIABLE POINT
- TO COMBAT THIS OUR ALLOCATION DOES NOT ACTUALLY FREE ANY BLOCKS SO NOT VALID FOR SUSTAINED USE

The Future of Python-Pin

- Better memory management
- Finish 32-bit support
- Instructions for Mac and Windows

Acknowledgements

Tyler Bohan
Kevin Chung
Dan Guido
Robert Meggs
Jonathan Salwan
Rich Smith
Paolo Soto
Alex Sotirov
Kai Zhong
baszerr.eu

Thanks for tuning in!

- Slides and pin tools will be posted to twitter, for real this time
  - @ancat/@1blankwall1