

THE WHOLE NINE YARDS

DEEPSEC 2012

INTROS

- Peter Morgan
 - Senior Consultant at Accuvant LABS, previously at Matasano Security.
- John Villamil
 - Senior Consultant at Matasano Security, previously at Mandiant.

BOTH

Fuzzing becomes really useful to us on a day to day basis

Most of the projects we work with require some sort of fuzzing

HISTORY OF MONKEYHERD

- We don't play defense... much; We're offensive
- This was driven by need
- What this most assuredly is not
- Voila! Monkeyherd

PETE

We are not defensive testers!

Through offensive testing we have learned some things that we think would help defensive testers!

We built the earlier iterations of this software to fulfill a testing need, then found it easily adaptable to further needs

Looking back, we haven't seen much discussion on the full lifecycle of implementing a fuzzing framework

What this is not:

- * How to write a fuzzer
- * Why dumb fuzzing works
- * A story about a dumb fuzzer that found OMG bugz!

FUZZING

Companies that do it well

- Microsoft
 - Microsoft runs fuzzing botnet, finds 1,800 Office bugs
 - Automated Penetration Testing with Whitebox Fuzzing
 - SAGE: whitebox fuzzing for security testing
 - Fuzz Testing at Microsoft and the Triage Process
 - <http://rise4fun.com/>
- Google
 - Fuzzing at Scale (<http://googleonlinesecurity.blogspot.co.at/2011/08/fuzzing-at-scale.html>)
 - Adobe admits Google fuzzing report led to 80 'code changes' in Flash Player
 - Fuzzing for Security (<http://blog.chromium.org/2012/04/fuzzing-for-security.html>)
- Adobe
 - Fuzzing Reader - Lessons Learned (http://blogs.adobe.com/asset/2009/12/fuzzing_reader_-_lessons_learned.html)

JOHN

WHY AREN'T THERE MORE?

Requirements for fuzzing

- Some basic knowledge
 - Understanding that fuzzing is beneficial
- The motivation to find and deal with bugs
- Company support
 - Time
 - Resources
 - Personnel
- Experience with the fuzzing process
 - This is covered by the talk



JOHN

CURRENT FRAMEWORKS

- The most popular are Peach and Sulley
 - They each support useful operations such as code coverage and target reboot
- The biggest disadvantage to using them is having to learn how they work.
 - Fuzzing needs to be flexible with a quick startup time
- Other fuzzers
 - Fusil
 - <https://bitbucket.org/haypo/fusil/wiki/Home>
 - Radamsa
 - <http://code.google.com/p/ouspg/wiki/Radamsa>
 - Zzuf
 - <http://caca.zoy.org/wiki/zzuf>

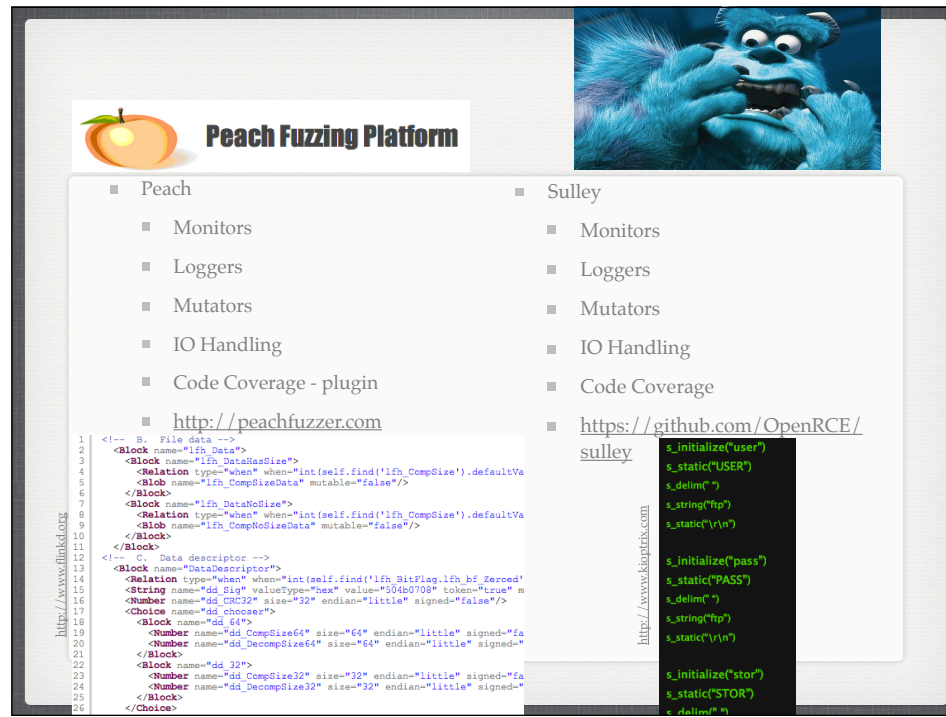
JOHN

Fusil has mangle.py, a very nice mangle library.

Radamsa is very easy to use.

Zzuf also supports code coverage information.

If you are going to use a premade fuzzer, see how it handles the input method for the application. For example, see how it handles packet fuzzing and state if the application accept network data.



JOHN

Both are great if you know the details of the input.

They both support major fuzzer features.

Peach uses an xml based template for the input types for describing a file

Sulley uses an API

THE STORY

Its not just about finding bugs to exploit

- Fault injection testing is the process of studying a program through its behavior with unintended input.
 - Crashes* are data which help construct a model
 - This data is used not only to fix/exploit bugs
 - It is used to optimize every step of the fuzzing process and gets updated with new software versions
- Simply relying on a current popular framework fails to use this data

PETE

* not just crashes; execution traces can be useful too

* allude to a case where instead of hunting for crashes, the framework is oriented to determine what inputs will allow a certain BB traversal

FOR WHOM?

- For builders:

- The ideal is dedicated testing nodes running on nightly builds
- Continually updated with samples to stress new code as the code is added
- Usually not possible - see "Fuzzing Requirements"
- The very minimum should be before a public release



WAIT FOR IT

.....Wait for it.....

PETE

This talk is targeted toward developers, product security teams, and admittedly, bughunters

- * Think Continuous Integration
- * Distributed
- * Trying to start this process the week of a release is probably not going to work
- * The true wins here come from integration
- * Build it into the SDL
- * Make devs aware their software will undergo fuzzing

ADVANTAGES OF A BUILDER

- Source code and intimate knowledge of how a program works
 - Sees incremental changes to a program over a period of time
 - Can create a large set of sample input for maximum code coverage



PETE

This saves valuable reversing time

Knowledge of the development teams
practices
internal motivations
corporate culture, etc

DIFFICULTIES FOR DEVELOPERS

- Not breakers; not just looking for one vuln, looking for ALL vulns
- Vulns->Bugs
- From the security perspective, the cards are stacked against you
- Large teams with async checkins
- Modern code shipping timelines :)
- Resources

PETE

Simple 5-line python fuzzers will not do

Randomized positive test case modulations to look for errant crashes that may allow exploitability is cherry picking, we need to be streetsweepers

* this works for offensive testers

* this doesn't help the defense

Developer code change

Ambulance bug hunting

Resources:

Money

Time

Expertise

Interest

FUZZ NODE PROCESS OVERVIEW

- Enumerate attack surface
- Pool of samples for code coverage
- Mutation/Generation
- Automated input delivery
- Grabbing crashes and exceptions
- Storing the data
- Run analysis on crash data

John

How do we know when there is enough samples?
What is code coverage useful for?
Data is used when analyzing crashes.

LETS DIVERGE

- Single-case fuzzing; this has been done before
- Vision of how this will work
- KISS
- Monkeyherd's features



PETE

Lead in to the distributed stuff

Vision of how this will work

Its really easy to get excited here:

We see a skynet-style fuzzing farm operating thousands of nodes from outer space scaling at will, autonomously based on a doctorate level heuristic with the ability to alert the devs when a serious issue is found

Hold on.

Lets remember Ben Nagy's great talk about fuzzing, keep is simple; don't over-engineer.

Think about this like a good vine gene, let it grow around the things it needs do, avoid over-engineering from the start

That being said, we should have some thought about how this will work

DISTRIBUTED TESTING

- Real-world defensive testing may need dozens to thousands of testing nodes for proper coverage
 - How can we know?
- Scalability should be considered at the start
- Inherent problem sets arise

PETE

Allude to code coverage

Optimization

* Don't worry about optimization yet, there might be time for that later, if there isn't you probably shouldn't be spending time on it here

CHALLENGES OF SCALING

- Node maturation
- Test case communication
- Avoiding duplicates [input,crash]
- Node status profiling
- Communicating results
- Optimizing behavior mid-cycle

PETE

CHALLENGE: NODE MATURATION

- Bare install -> functional testing node
 - Communication channel
 - Software installation
 - Tool delivery

PETE

BASIC NODE MATURATION

- toolchain installation
- fuzzer software deployment
- master node check-in
- where will this be?

PETE

- * installation
 - puppet
 - shell scripts
 - cfengine
- * software deployment
 - similar to above
 - git checkout
- * scripted to operate successfully against an environment of choice
- * here one should think about internal vs. external hosted nodes
 - internet connected?
 - net environment
- * Internal: install from network share/ local repo

MONKEYHERD DESIGN DECISIONS

- Human interaction required
- Built for operation on EC2
- SSH
- Git
- Ruby/Python

PETE

EC2

Why?

you may be tempted to try to use random hosts for this task
avoid the pitfalls of trying to debug this across a dozen OS/version combos

Pick something that allows consistency; we will revisit pitfalls later

SSH

alternative is spiped
obviously need secure comm channel
establish tunneling to master nodes

Git

could be any VCS,
you want to be able to quickly hop to a fuzzer node and have an idea of what rev its running

Ruby

pick any instrumentation language, ruby is my fav, John likes python
monkeyherd is interesting in that it doesn't matter!

CHALLENGE: TEST CASE COMMUNICATION

- Design decision: generate and send, or build on node?
- How?

PETE

What should we consider?

- * file size issues are obviously a problem
 - * small file-format fuzzing vs movie files for media players
- * tracking of fuzz test case data
 - * how to do that?
 - * imagine when the test case causes a valuable crash
 - * John will get back to that later

We will need a C&C for this

COMMAND AND CONTROL

- Problem sets share a common need for C&C
- Tons of options
 - Web services
 - REST framework
 - DSL
- KISS and REDIS

PETE

Which problem sets?

- * test case distribution
- * actual command and control
- * status requests
- * results transmission
- * GUI automated sync

This will be insanely useful in the future, as in any distributed system

There are literally tons of options

- * any message queue
- * Web services
- * HTTP with REST
- * Custom DSL

Before you spend time overengineering this too (starting to see a trend here? trust me it gets worse)

Go back. Keep It Simple.

Redis is a fast KV store
C with no deps outside libc
Built-in pub/sub

MONKEYHERD: REDIS

- KV store with simple data structuring operations
- More than just a C&C
- Master and slaves communicate through Redis node
- C&C setup using LIST operations
 - Not PUB/SUB
- PROTIP: ~~Windows~~

PETE

Obviously a security hazard, ensure your node maturation phase takes into account the issues of someone taking over your C&C

DDOS is fun

More than a C&C

Could do so using PUB/SUB mechanisms, but in practice timing issues were encountered
LIST operations are persistent

PROTIP: Don't instrument on windows

C&C MESSAGES

- global_nodelist - SET - global list of all available nodes
- last_nodelist - SET - list of all responding nodes
- notifypub - PUBLISH - all slave nodes SUBSCRIBE to notifypub to listen for notification messages
- NodeID:CC:pause - LIST - Used to command node to pause operations
- NodeID:CC:crash - LIST - Set when debugging instance detects crash
- NodeID:crashlist - LIST of crash instance IDs - incremented
- NodeID:Crash:ID:doutput - debugger report of crash ID
- NodeID:Crash:ID:input - file triggering crash
- NodeID:Crash:ID:input_hash - md5 of input file

PETE

CHALLENGE: NODE STATUS

- Are nodes responding? How can we check efficiently?
- Simple PUB/SUB in Redis
- Reality: there is hand-holding needed

PETE

Ends up being 20 lines of ruby to broadcast 3 messages, check for responses, and list available/unavailable nodes in redis console

CHALLENGE: RESULTS COMMUNICATION

- Mostly Solved :)
- Put the results in redis directly
- Solved in other ways

PETE

An interesting issue is when input file is huge

- take a binary diff
- store the diff and the hash of the input file

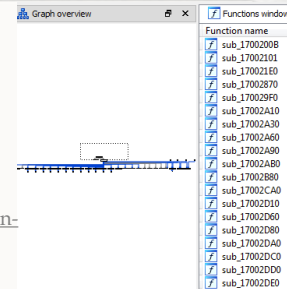
CODE COVERAGE

- Breakpoints

- IDA for function offsets
- pydbg/ragweed
- http://paimci.googlecode.com/svn/trunk/utis/code_coverage.py
- <https://www.corelan.be/index.php/2010/10/20/in-memory-fuzzing/>

- Dynamic Binary Instrumentation

- For on the fly checking of basic blocks
- In this talk we use PIN



```
count++      sub  $0xff, %edx
count++      cmp   %esi, %edx
count++      jle   <L1>
count++      mov   $0x1, %edi
count++      add   $0x10, %eax
```

JOHN

Get a list of functions from IDA and set breakpoints on them through pydbg/ragweed.

When a breakpoint is hit, remove that from the list. If new samples don't hit breakpoints, discard them.

PIN gives more detailed control.

Both are valid options and have their positives and negatives.

IDA may not find all the functions addresses and so the coverage breakpoints won't be as complete.

When using PIN, the initial application setup functions can be discarded when considering code coverage.

CODE COVERAGE: HOW DOES IT WORK?

- We use PIN by Intel
 - Dynamic Binary Instrumentation tool which interweaves your code with the program
 - www.pintool.org for more info. Documentation is excellent.
- What to record - choose one
 - Basic Block entrance or exit
 - Control flow instructions (jmp, ret, call)
 - Arbitrary instruction or function call (eg. coverage of malloc)

```
VOID LEVEL_PINCLIENT::BBL_InsertCall( BBL    bbl,  
                                     IPPOINT action,  
                                     AFUNPTR funptr,  
                                     ...  
                                     )
```

Insert call relative to a bbl.

Parameters:

bbl BBL to instrument
action Specifies before, after, etc.
IPPOINT_BEFORE is always valid for all instructions.
IPPOINT_AFTER is valid only when a fall-through exists (i.e. Calls and unconditional branches will fail).
IPPOINT_ANYWHERE will put the instrumentation at a place inside the bbl for best performance
IPPOINT_TAKEN_BRANCH is invalid for non-branches.
funptr Analysis function to call
... *IARG_TYPE*. Arguments to pass to funptr

JOHN

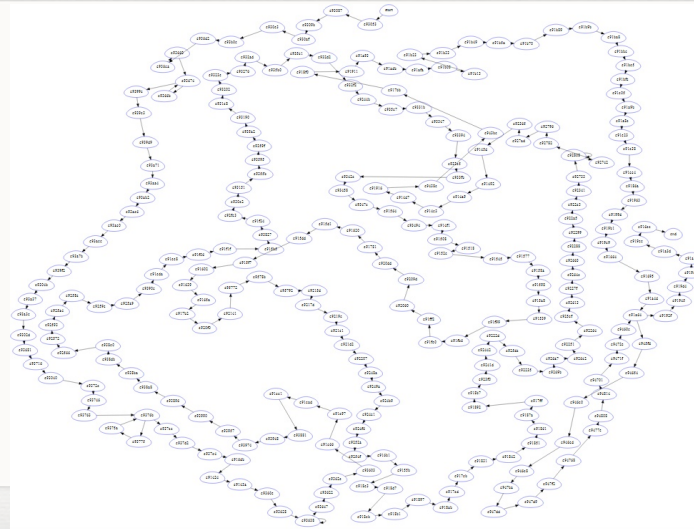
Instrumentation tools allow you to insert your code into a programs execution flow.

This code can be used to analyze or modify a program at run time.

Because PIN is dynamic, a PIN block is different from a regular basic block you will see in IDA. A PIN bbl is a single entry single exit piece of code. A regular bb has one entry and one exit but can contain calls to other functions within it.

Screenshot is of PIN API used to call a user defined function before each basic block.

NOTEPAD.EXE



JOHN

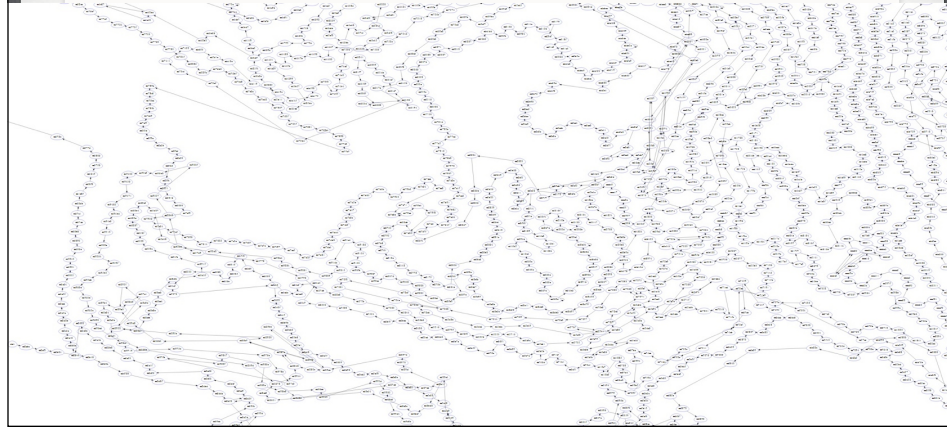
Using PIN it is easy to record any type of application data and graph it.

Screenshot is of control flow branches and calls in notepad.exe.

CALC.EXE

Each point is a branch found by PIN

Size difference ~650k

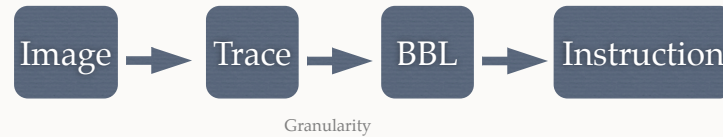


JOHN

A bigger amount of instructions increases

CODE COVERAGE: HOW DOES IT WORK?

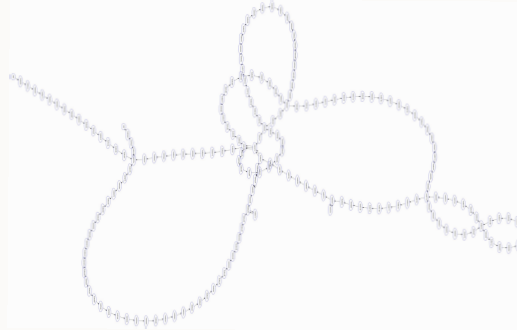
- PIN allows us to base coverage on additional information
 - Stack data
 - register values
 - Caller instead of callee
 - Number of instructions executed before break
 - etc
- Additional flexibility allows for more detailed data



JOHN

TAINT TRACING

- Based on simple rules
 - track registers and memory accesses
 - if source of an operation is tainted, the destination becomes tainted
- Implementations:
 - BitBlaze
 - Dytan
 - PrivacyScope
 - libdft
 - Minemu
- add eax, ebx
 - eax
- xor eax, eax
 - eax



JOHN

TAINT PROPAGATION

- What is tainted at crash time?
- Rules checking if tainted data is passed into system functions. ie strncpy(dest, src, tainted length)
- Made Easy with PIN
 - INS_OperandIsMemory, INS_OperandIsREG, INS_OperandIsImmediate
 - Usually done using the XED2 engine and function pointers

```
...
op[XED_ICLASS_ADD] = &op_add;
op[XED_ICLASS_LEA] = &op_lea;
op[XED_ICLASS_MOV] = &op_mov;
op[XED_ICLASS_POP] = &op_pop;
op[XED_ICLASS_PUSH] = &op_push;
op[XED_ICLASS_SUB] = &op_sub;
op[XED_ICLASS_XCHG] = &op_xchg;
op[XED_ICLASS_XOR] = &op_xor;
...
```

OPCODE LEVEL_CORE::INS_Opcode(INS *ins*)

On ia-32 and Intel64 the opcodes are constants of the form XED_ICLASS_name. The full list is distributed as part of the Pin kit), and the enum definitions are in the file "xed-class-enums.h".

Use **INS_Mnemonic** if you want a string.

Returns:

Opcode of instruction

JOHN

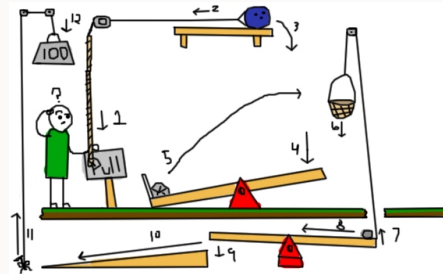
TAINT PROPAGATION

- How can you check dozens or hundreds of allocated memory chunks?
- Shadow memory techniques help to solve memory propagation
 - + Extensible and fast with proper optimizations
 - - Can use a lot of memory
 - Each bit of the shadowed byte can be used to record information
 - Has this byte been freed
 - Is the next/previous byte tainted
 - How many times has this byte been accessed so far

JOHN

AUTOMATION

- Clearly, a huge thing for fuzzing
- Already mostly done
 - CLI
 - Network
 - GUI



PETE

The obvious advantage of a fuzzer is the automated testing of input payloads, without sufficient automation it doesn't have a place.

Audience involvement:

Who has written a fuzzer before?

Who has run into a GUI app that they wanted to fuzz, but didn't due to complexity?

We have, lots.

Simple axiom: if its not easy to hit, its not been hit hard enough

One of Monkeyherd's advantages is the tight coupling with GUI automation

AUTOIT

- Windows application
- Free to use
- Ruby gem
 - FFI to DLL
- Easy to use functions:
 - coordinate based click
 - window title accessors
 - keystroke automation
 - sleep



<http://www.autoitscript.com/site/autoit/>

PETE

LESSONS LEARNED IN GUI AUTOMATION

- Pete's Razor: $\text{sleep}(x * 3)$
- The Mouse Is Trying To Kill You
 - Don't click unless you have to!
 - Be vigilant of where the pointer is
- Prune directory used with file dialogs
- Navigate to absolute paths in file dialogs
- Network is easier, C&C is a huge win here:



PETE

Whenever you think you need a sleep of X, you want $x * 3$
This is a stern suggestion for things like file dialogs

PROTIP: keep directories from which you're accessing things mostly clear, it will reduce dialog population time which can be considerable

Always navigate to absolute paths in file dialogs

Where the pointer is:

Burned a solid night debugging errors that were coming up because I didn't reposition the mouse cursor to a safe position before continuing automation

Don't forget calculating pixel positions of things on the screen always will depend on the display resolution. It's almost always better to find the right combination of tab, and arrow keys.

Network fuzzing:

- Prepare the test case message
- Drive the application to the state where it will accept a testcase message
- Alert the test case handler to fire the message
- Reset state

GUI AUTOMATION PROCESS

- Record the workflow by hand
- Decompose application usage into states
- Use Keystroke automation first!
- If workflow requires, implement mouse clicks
 - After each use, reposition cursor in safe area
 - Use GUI-based assertions
- Observe places to pause other components
- Test now, bask in bug glory later

PETE

Mock up a file-format case:

- * Open application – record time
 - * Invoke the File | Open dialog – record time
 - * Navigate to an absolute reference point, then navigate to relative file
 - * helps to configure the target to have a trivially reached directory to populate
- with testcases
- * Launch a positive testcase
 - * Revert to the File | Open process and repeat
-
- * Now launch a negative test case
 - * Observe the debugger

GUI ASSERTIONS?!

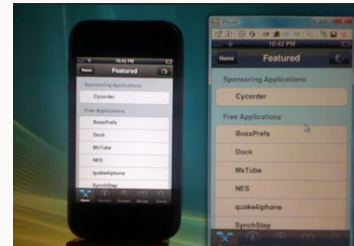
- These are minor tests that aim to check the state of the GUI automation phase
- Helpful when they scale linearly with the amount of GUI automation required to instrument the app
- AutoIt
 - Window Titling
 - Positional color tests

PETE

- * use these to assert that the GUI automation is in the right state
- * you want to link this with the C&C to ensure you can control the state
- * instrument a kill_harness command that will reset the target app to a known state

TRICKS WITH OTHER OSes

- Might seem like cheating, but VNC!
- The window to the world?
 - iOS
 - Android
 - OSX
 - Linux



PETE

NOT ALL VULNS ARE CRASHES

- Detach from the need to find “crashes”
- Memory safety bugs are great, but so are logic bugs
- “This is the weapon of the enlightened few. Not as clumsy or random as a memory overwrite. An elegant weapon for a more civilized age.” -Not Ben Kenobi

How?



Logic vulns can be more subtle, and sneaky than memory safety bugs

Think of cases where the goal of fuzz testing isn't simply crashes, but attempts to arrive at a location in the binary.

Simply watching for crashes is insufficient

How?

Use breakpoints to target sensitive or critical application functions to assert if execution arrives at the sensitive areas.

!EXPLOITABLE

- Most useful feature is the hashing
 - Simple algo that hashes major and minor stack frames
 - Its not perfect - duplicate crashes can have a different hash
 - easily portable to gdb
- cdb on windows
 - -g -G -o -kqm
 - -logo to log stuff
 - If exception: -c "\$<filename" will run !exploitable and quit
 - extract classification and hash and add to directory tree

```
1 {730,ea00}: Access violation - code c0000005 (first chance)
2 First chance exceptions are reported before any exception handling.
3 This exception may be expected and handled.
4 eax=00007700 ebx=04f293f0 ecx=fffffa00 edx=0d05a000 esi=00000000
5 eip=676b585a esp=057bf07c ebp=00005690 iopl=0         nv ui
6 cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010216
7 *** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
8 Files\VideoLAN\VLC\plugins\audio_filter\liba52tofloat32_plugin.dll -
9 liba52tofloat32_plugin\vc_entry_license_1_2_0!+0x436a:
10 676b585a 8b0a          mov     ecx,dword ptr [edx]  ds:0023:0516c000=????????
11 0:012> $<dbgcomm.txt
12 0:012> !load winext\msec.dll
13 0:012> !exploitable
14 *** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll -
15 Exploitability Classification: UNKNOWN
16 Recommended Bug Title: Read Access Violation starting at
17 liba52tofloat32_plugin\vc_entry_license_1_2_0!+0x0000000000000436a (Hash=0x6453581b,0x64185860)
18 0:012> qq
19 quit:
```

SO YOU HAVE A CRASH

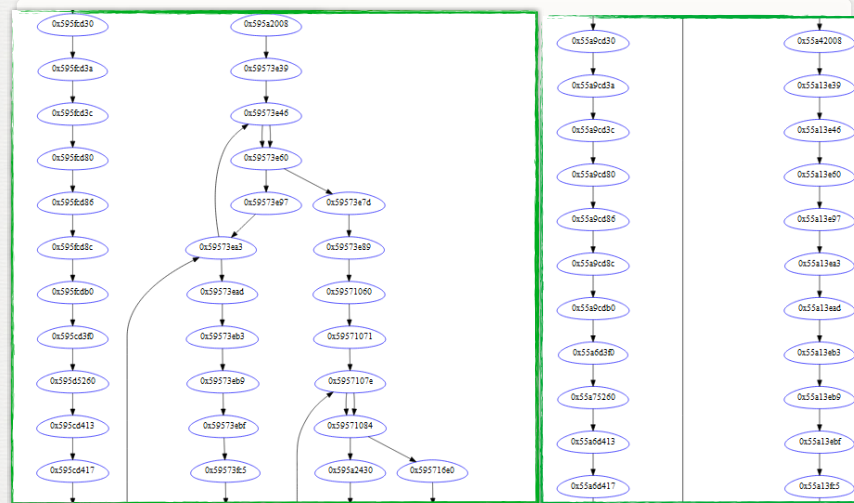
- What to do?
 - fix the bug
 - decide if it is a security threat
- Speeding up the process
 - Diffs between good and bad input
 - Code graphs showing execution flow
 - Specific functions per bug type (ie mallocs/frees)
- Record any app or bug specific information



JOHN

Analysis

VISUAL DIFFERENCES



JOHN

Visual differences highlight locations where input has made the program divert from its routine.

In the cases of a crash, this diversion is unexpected

The tainted data in the first different code block may help in figuring out why unpredicted behavior has occurred.

SOLVING FOR CODE COVERAGE

- What happens when a system function is called?
 - What happens to the taint?
 - Automate taint tracing of system functions - still need to manually define the argument types.
 - Hardcode common functions into your taint propagation logic.

```
anybreak@ubuntu:~/Desktop/bap-0.64$ ./utils/topredicate -g -il simp.il -stp-out /tmp/f -post goal -solve
WARNING (Typecheck): infer_ast -check option is deprecated. Please use typecheck_expression instead.
Hecks: Removing backedges
SSA: Translating to SSA
SSA: Creating new cfg
SSA: Computing defsites
SSA: Computing dominators
SSA: Adding phis
SSA: Added 1 phi
SSA: Grouping phis by block
SSA: Translating block BB_Entry
SSA: translating stats
SSA: going on to children
SSA: Translating block BB_2
SSA: translating stats
SSA: going on to children
SSA: Translating block BB_4
SSA: translating stats
SSA: going on to children
SSA: Translating block BB_Exit
SSA: translating stats
SSA: going on to children
SSA: Adding 1 phi to the CFG
SSA: Done translating to SSA
Use Simp: Starting iteration 0
SSA: rm_phis: removing phi for goal_BB:bool
SSA: rm_phis: adding assignments for goal_BB:bool
Model:
0 EBX 6 01 -> 0x22
```

BAP is awesome!

JOHN

BAP

- Binary Analysis Platform
 - Transforms into an intermediate language
 - Optimizes the intermediate language
 - Solves to satisfy a “verification condition” upon that optimized and simplified intermediate language
 - We use a custom taint tracing tool

unoptimized BAP IL

```
addr 0x0 @asm "mov    $0x20,%eax"
label pc 0x0
R_EAX:u32 = 0x20:u32
addr 0x5 @asm "mov    $0x2,%ecx"
label pc 0x5
R_ECX:u32 = 2:u32
addr 0xa @asm "xor    %ecx,%ebx"
label pc 0xa
R_EBX:u32 = R_EBX:u32 ^ R_ECX:u32
R_OF:bool = false
R_CF:bool = false
R_AF:bool = unknown "AF is undefined after xor":bool
R_PF:bool =
~low:bool(R_EBX:u32 >> 7:u32 ^ R_EBX:u32 >> 6:u32 ^ R_EBX:u32 >> 5:u32 ^
R_EBX:u32 >> 4:u32 ^ R_EBX:u32 >> 3:u32 ^ R_EBX:u32 >> 2:u32 ^
R_EBX:u32 >> 1:u32 ^ R_EBX:u32)
R_SF:bool = high:bool(R_EBX:u32)
R_ZF:bool = 0:u32 == R_EBX:u32
```

JOHN

BAP is a multiplatform suite of tools that operate on assembly instructions.

It translates intel instructions into their explicit operations, spelling out what each operation does.

If you think about a push instruction, it will sub esp and mov a value into esp.

SOLVING FOR CODE COVERAGE CONT.

- Through solvers we can...
 - Cut down on the number of samples
 - Automate how to hit new code
 - Pass constraint solution to fuzzing nodes and concentrate efforts per code path (per constraint)
 - Pass along tainted instructions and have BAP convert to IL and solve for a specific path
 - Fuzz only tainted input within each targeted set of basic blocks?

JOHN

RELEASE

- Planning to release Monkeyherd Q12013
- Feedback

REFERENCES

- “How to Shadow Every Byte of Memory Used by a Program”, <http://valgrind.org/docs/shadow-memory2007.pdf>
- “BAP: The Next-Generation Binary Analysis Platform”, <http://bap.ece.cmu.edu/>
- Ben Nagy on Fuzzing, <http://seclists.org/dailydave/2010/q4/47>
- A Million Data Watchpoints, <http://www.dynamorio.org/pubs/zhao-million-watchpoints-CC08.pdf>
- Taint tracking in fuzzing, <http://cansecwest.com/csw11/Metrics%20for%20Targeted%20Fuzzing%20-%20Duran,%20Miller%20&%20Weston.pptx>

THANK YOU AND Q/A

- Contact Us:
 - Peter Morgan (@rockdon)
 - peterjmorgan@gmail.com
 - John Vilamill (@day6reak)
 - johnvillamil2010@gmail.com