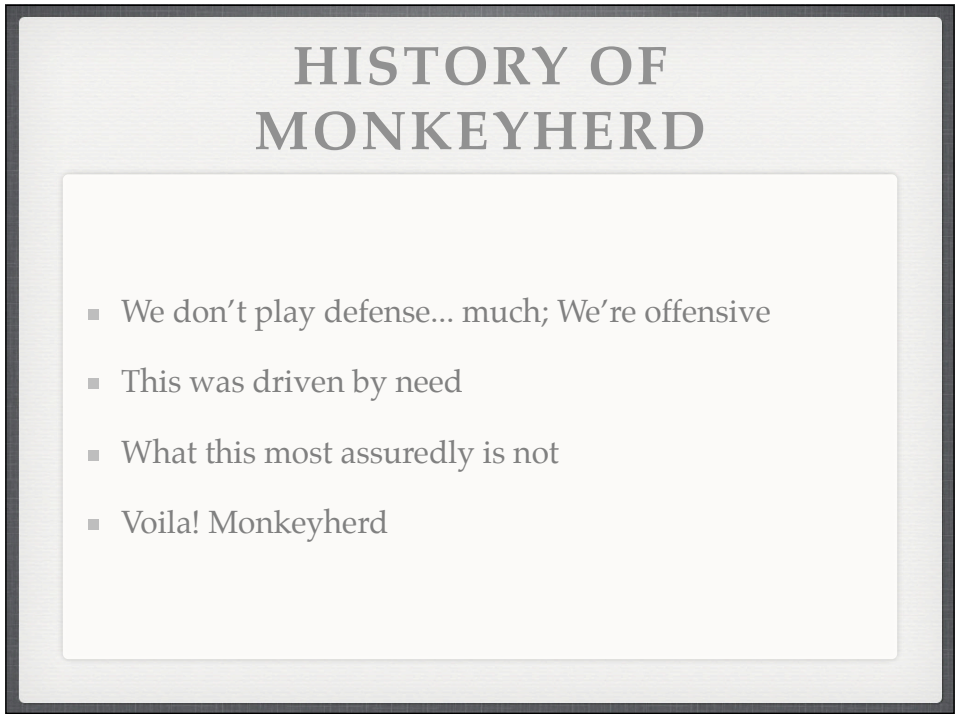
THE WHOLE NINE YARDS

DEEPSEC 2012

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



PETE

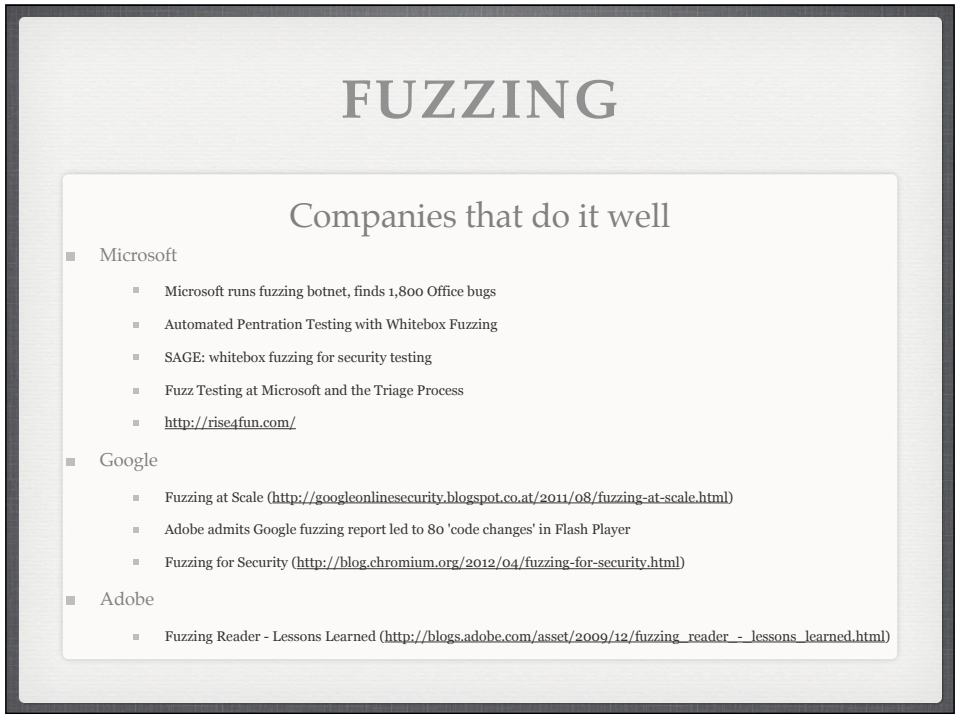
We are not defensive testers!

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 Pentration 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)

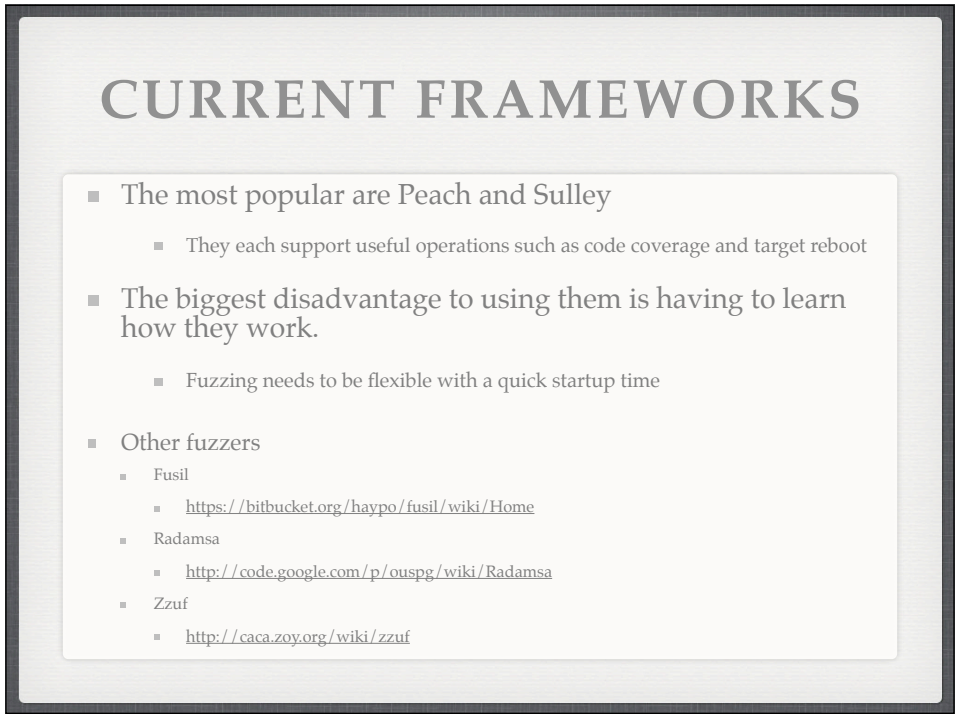
JOHN

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



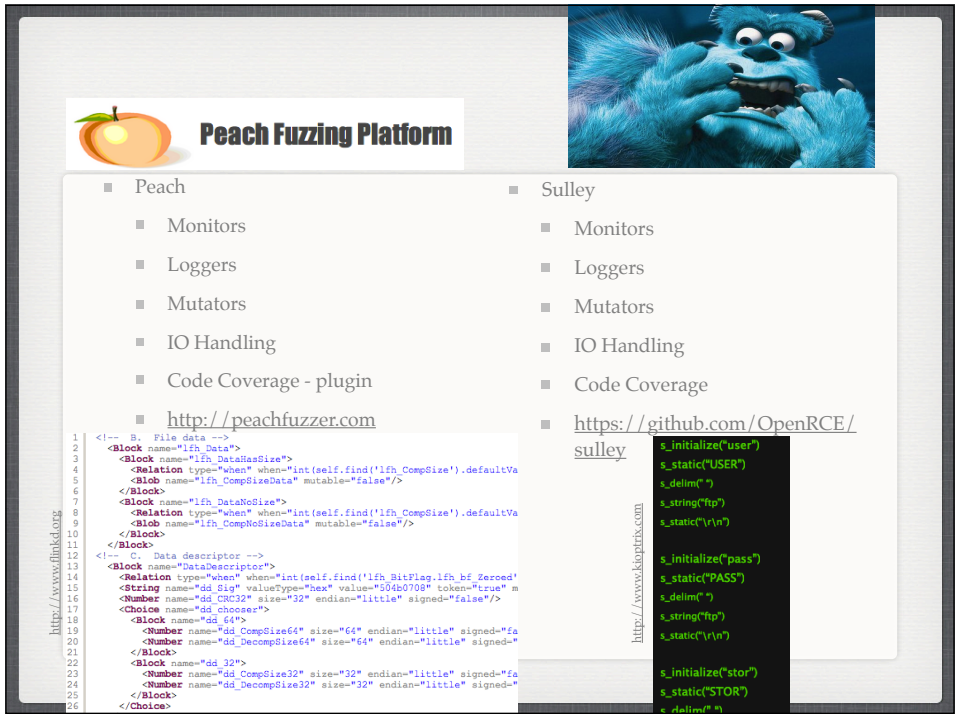
JOHN

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.



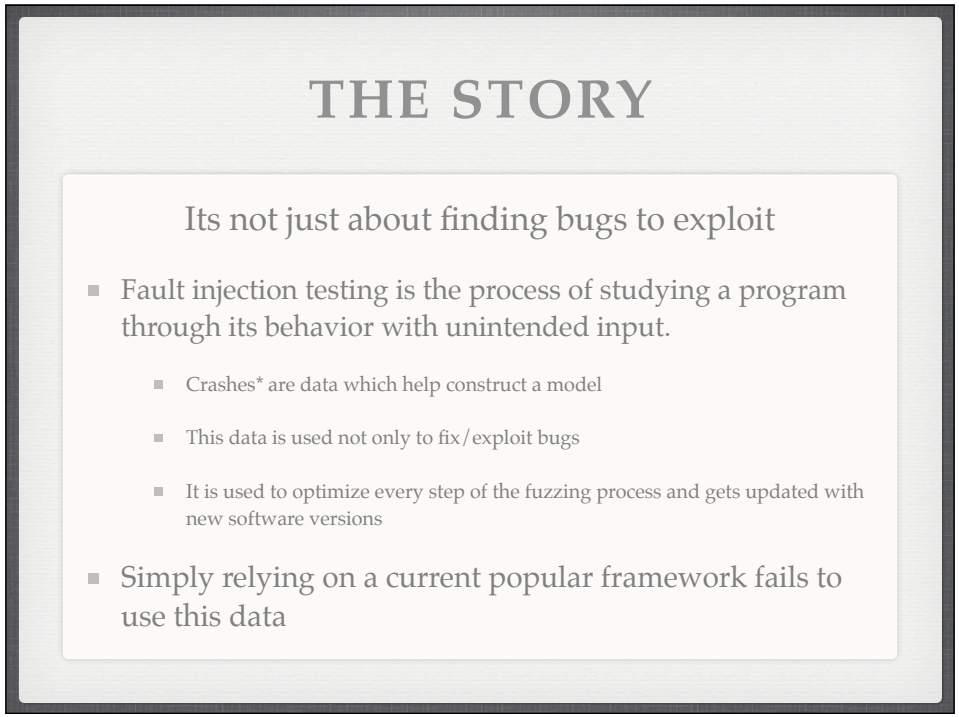
JOHN

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

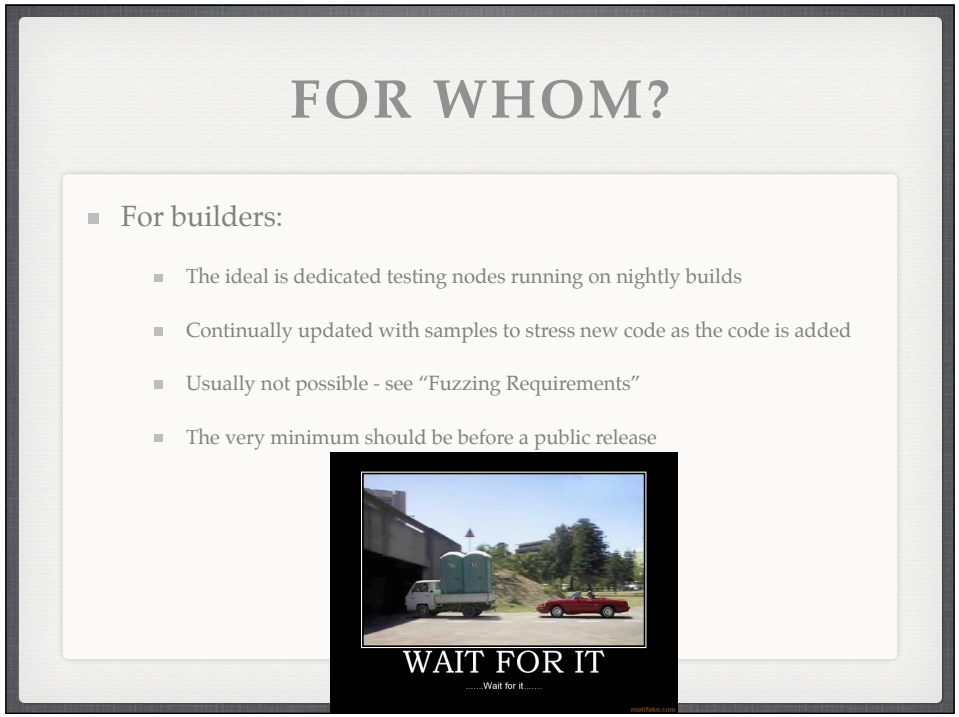
Sulley uses an API



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



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

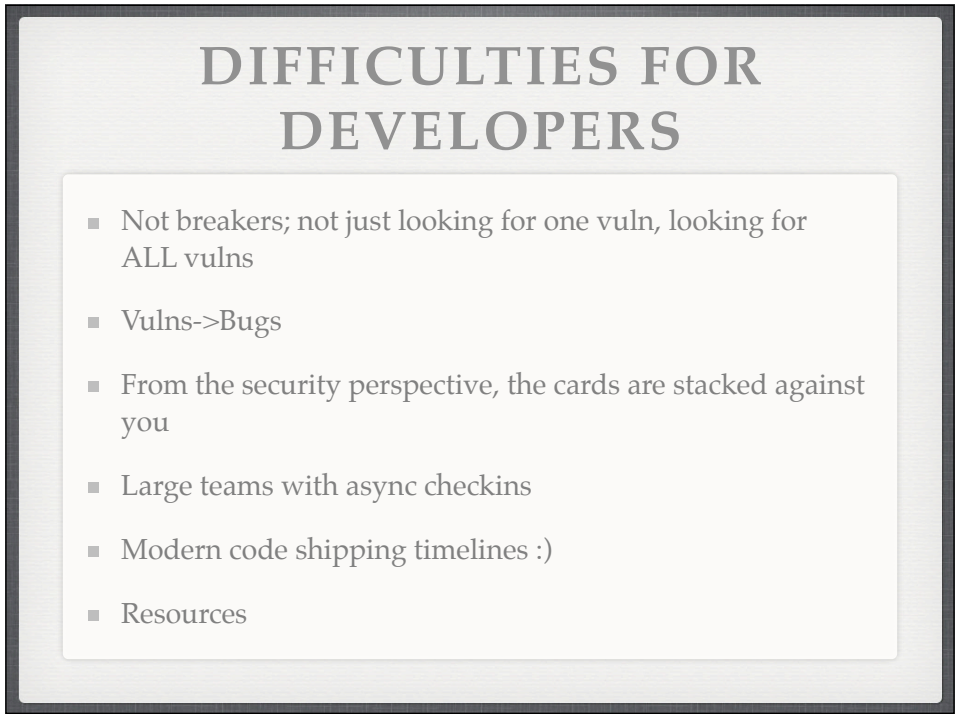
ADVANTAGES OF A BUILDER

- 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



PETE

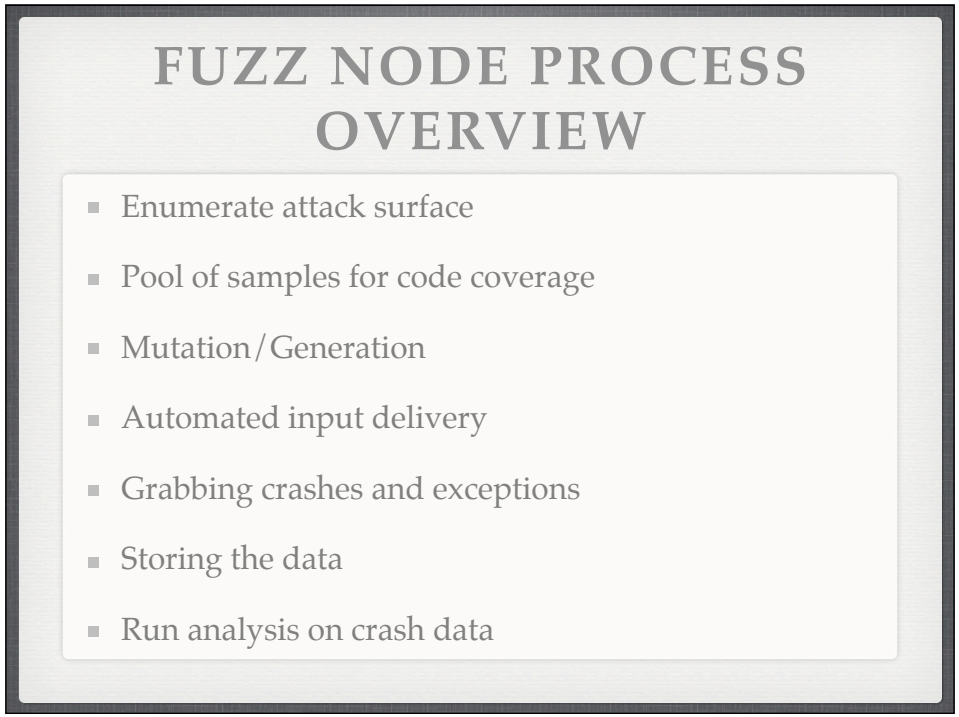
Simple 5-line python fuzzers will not do

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

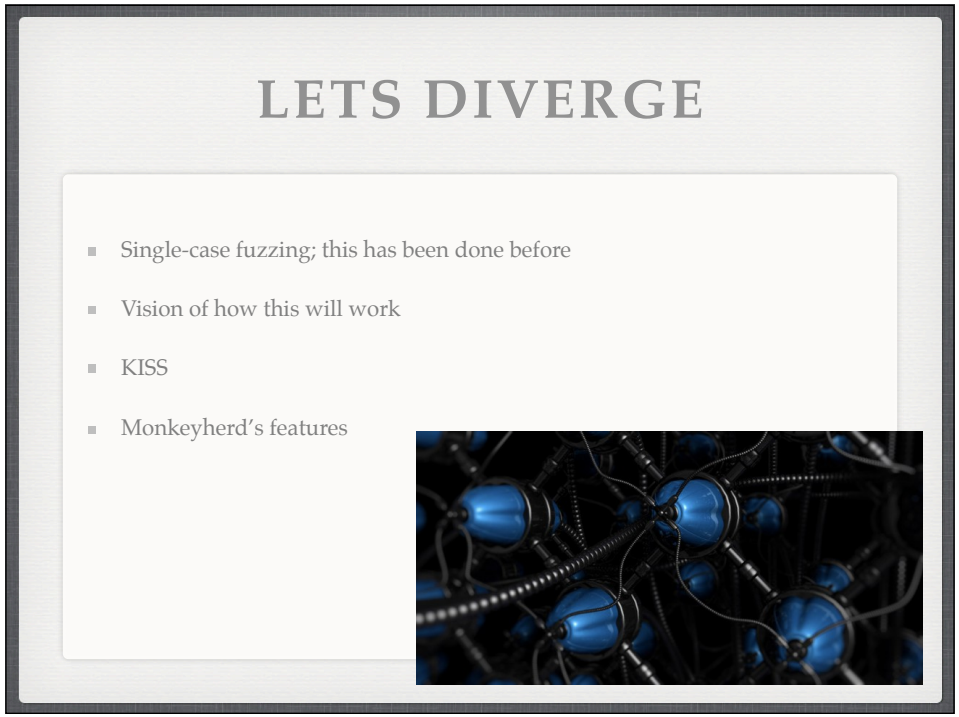
Ambulance bug hunting

Resources:
- Money
- Time
- Expertise
- Interest



John

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



PETE

Lead in to the distributed stuff

Vision of how this will work

Its really easy to get excited here:

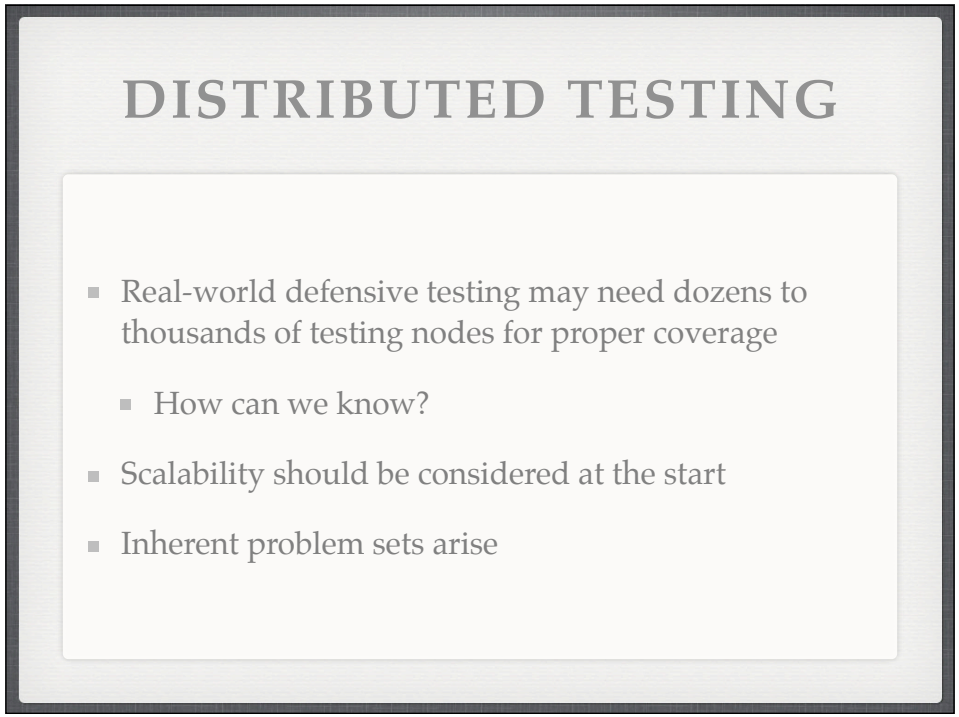
We see a skynet-style fuzzing farm operating thousands of nodes from outer space scaling at will, autonomously based on a doctorate level heuristic with the ability to alert the devs when a serious issue is found

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



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

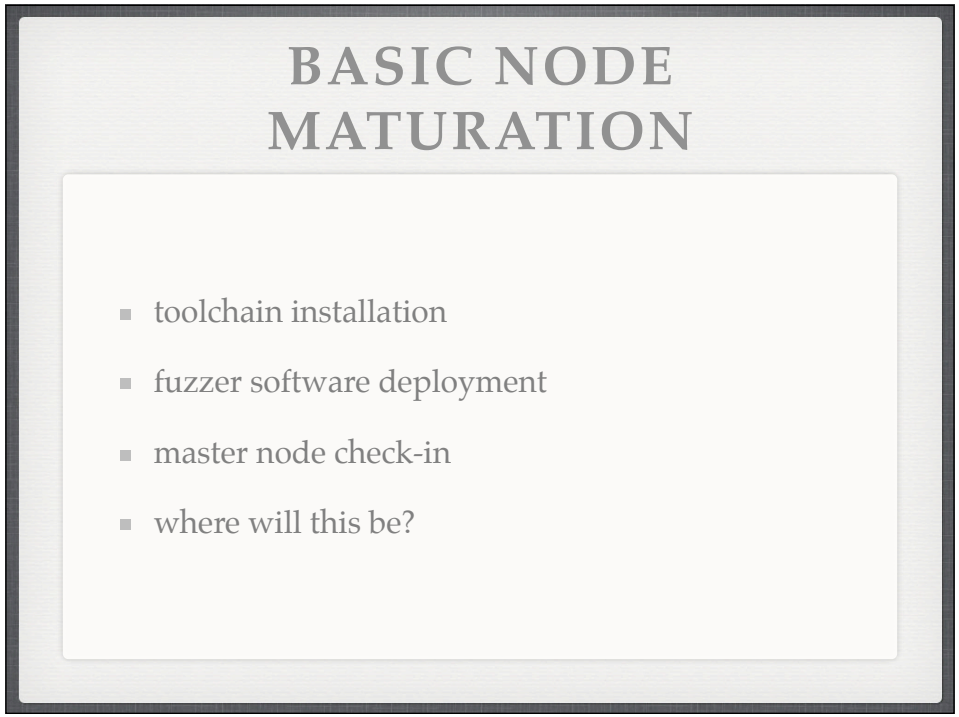
- Node maturation
- Test case communication
- Avoiding duplicates [input,crash]
- Node status profiling
- Communicating results
- Optimizing behavior mid-cycle

PETE

CHALLENGE: NODE MATURATION

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

PETE



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

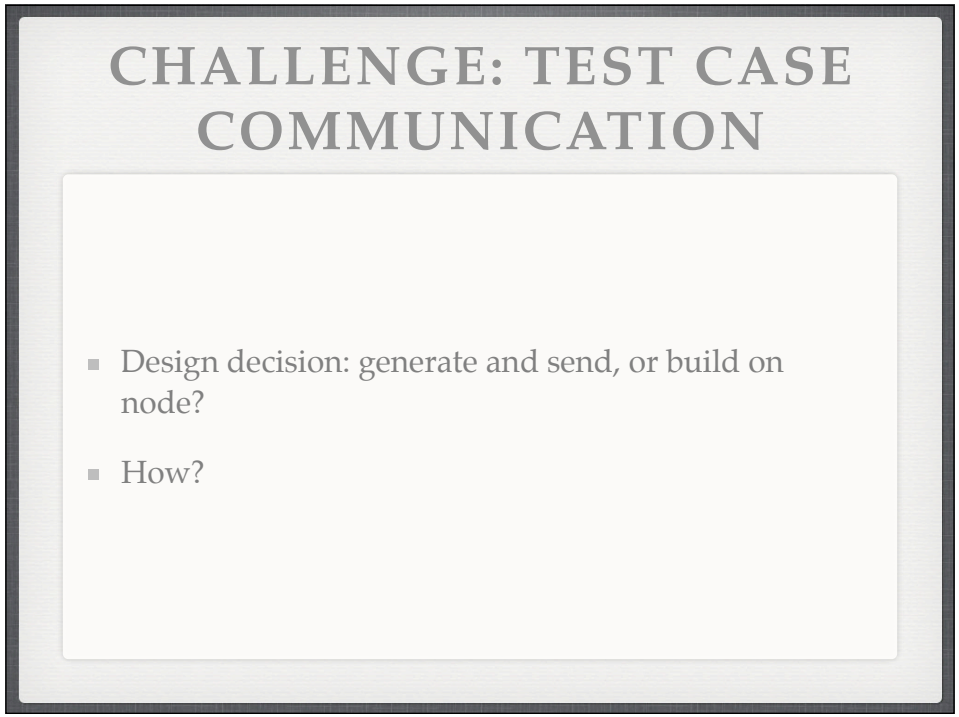
EC2
Why?
    you may be tempted to try to use random hosts for this task
    avoid the pitfalls of trying to debug this across a dozen OS/version combos

    Pick something that allows consistency; we will revisited 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!

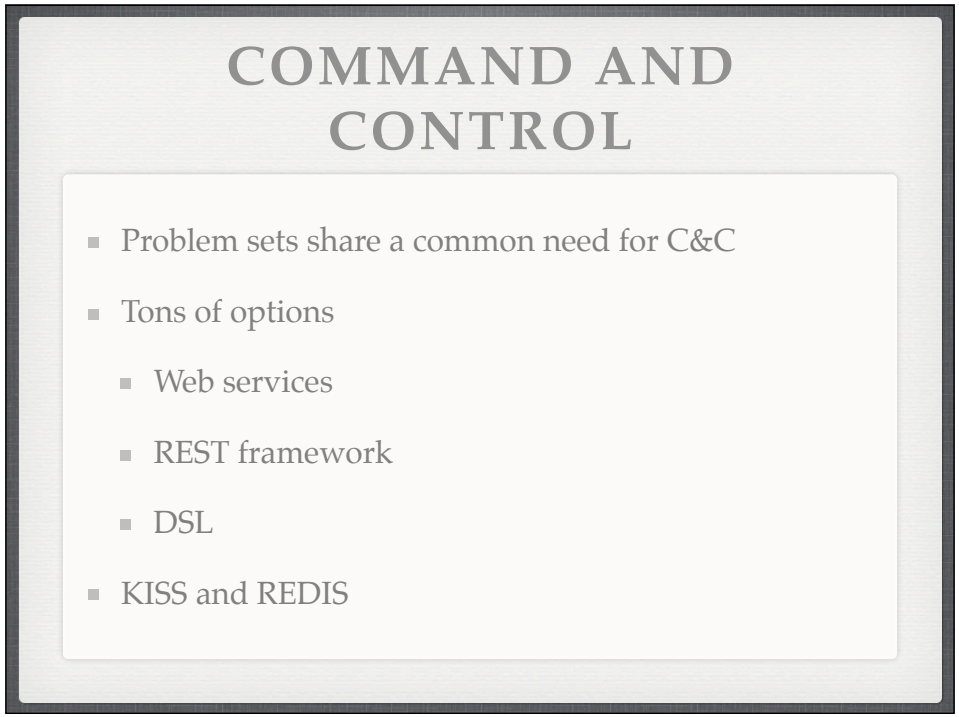


PETE

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



PETE

Which problem sets?
* test case distribution
* actual command and control
* status requests
* results transmission
* GUI automated sync

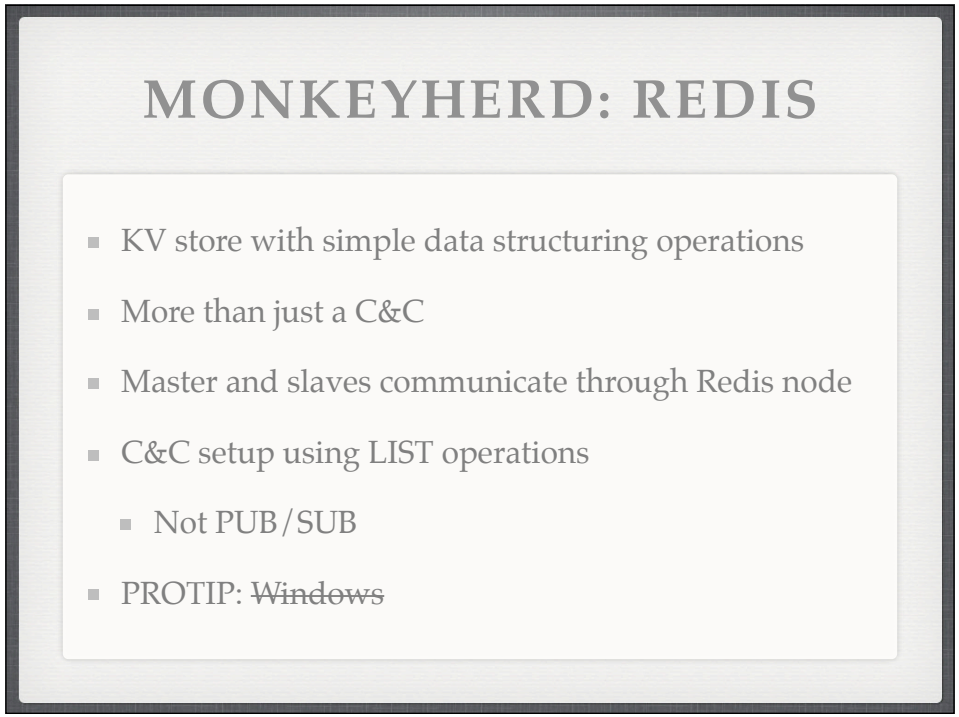
This will be insanely useful in the future, as in any distributed system

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



PETE

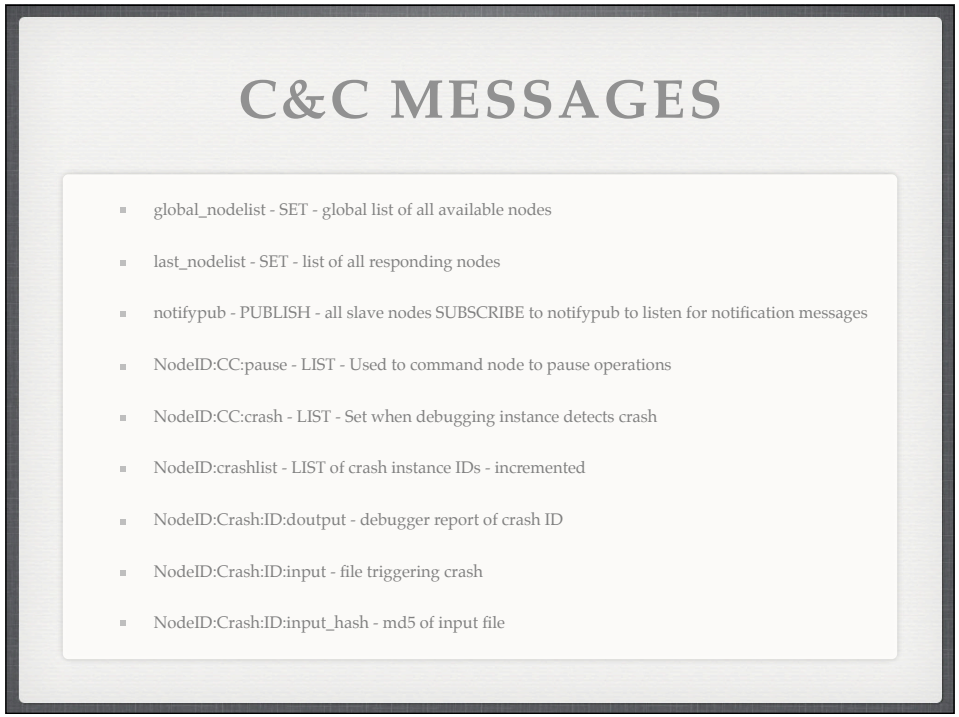
Obviously a security hazard, ensure your node maturation phase takes into account the issues of someone taking over your C&C

DDOS is fun

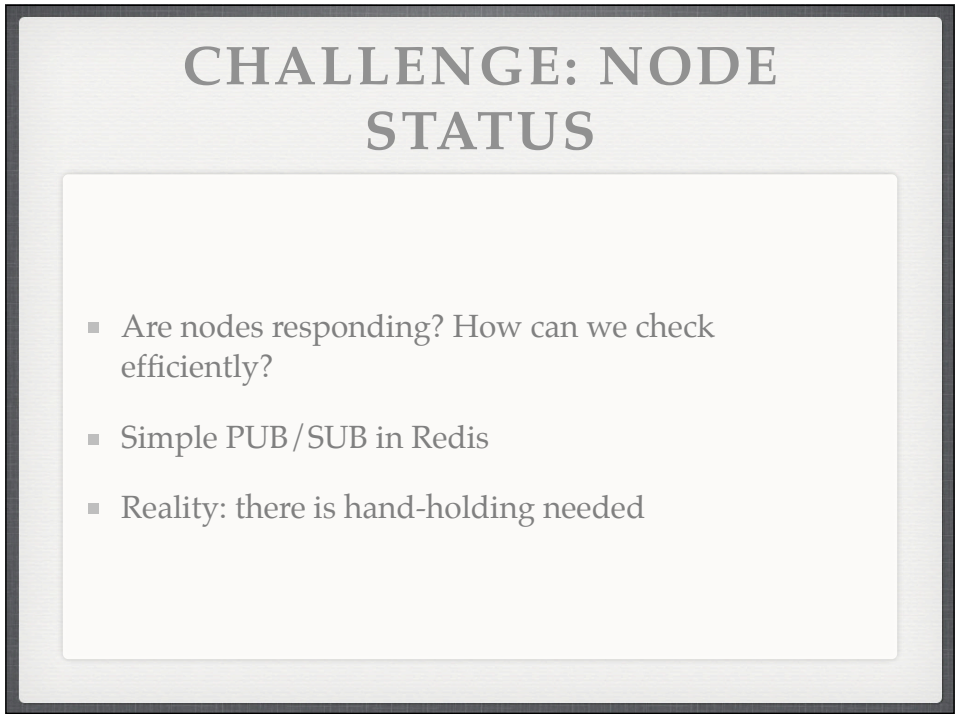
More than a C&C

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

PROTIP: Don't instrument on windows

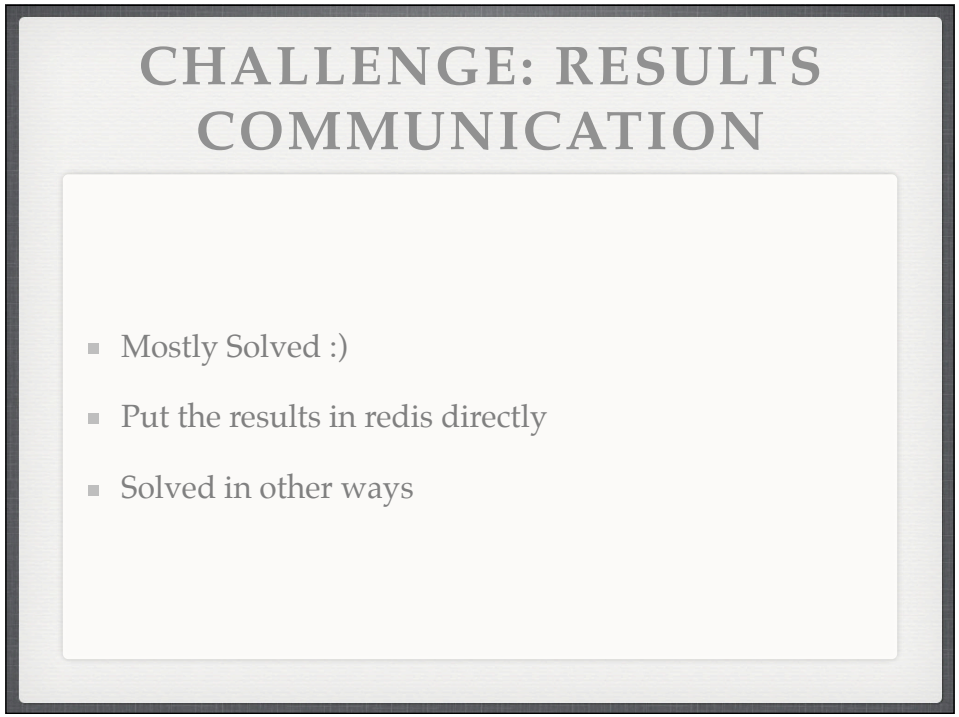

C&C MESSAGES
global_nodelist - SET - global list of all available nodes
last_nodelist - SET - list of all responding nodes
notifypub - PUBLISH - all slave nodes SUBSCRIBE to notifypub to listen for notification messages
NodeID:CC:pause - LIST - Used to command node to pause operations
NodeID:CC:crash - 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



PETE

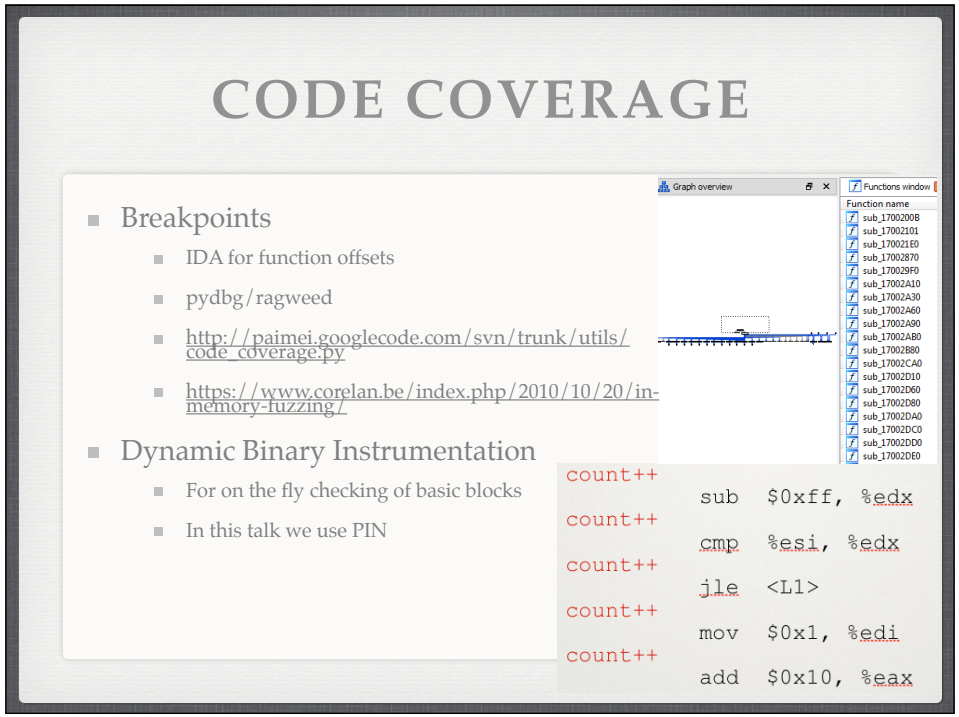
Ends up being 20 lines of ruby to broadcast 3 messages, check for responses, and list available/unavailable nodes in redis console



PETE

An interesting issue is when input file is huge

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



JOHN

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

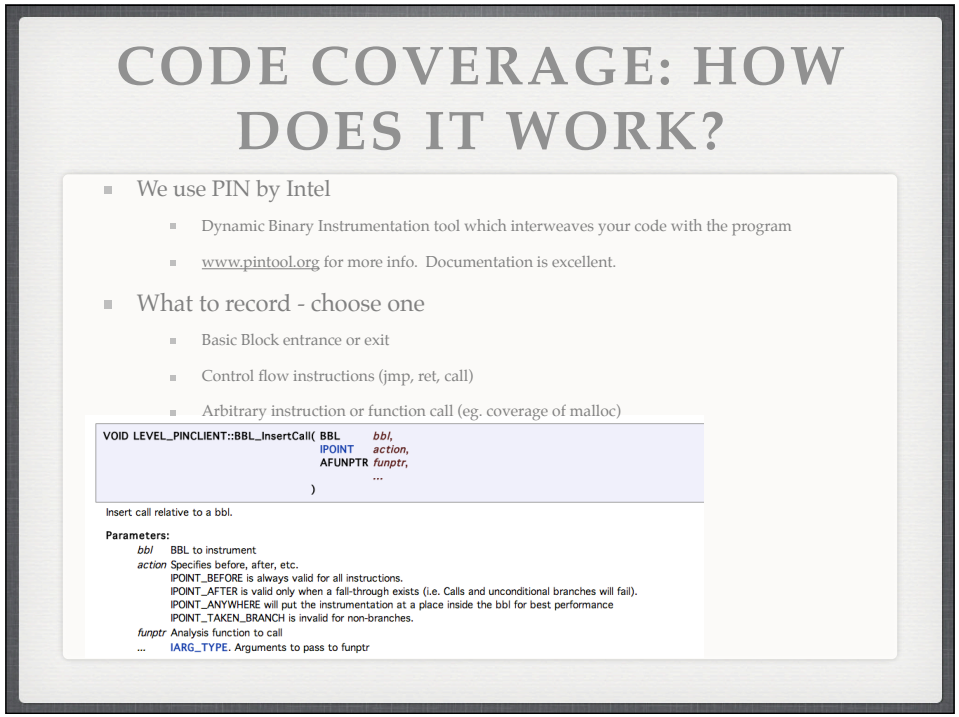
When a breakpoint is hit, remove that from the list. If new samples dont hit breakpoints, discard them.

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 wont be as complete.

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



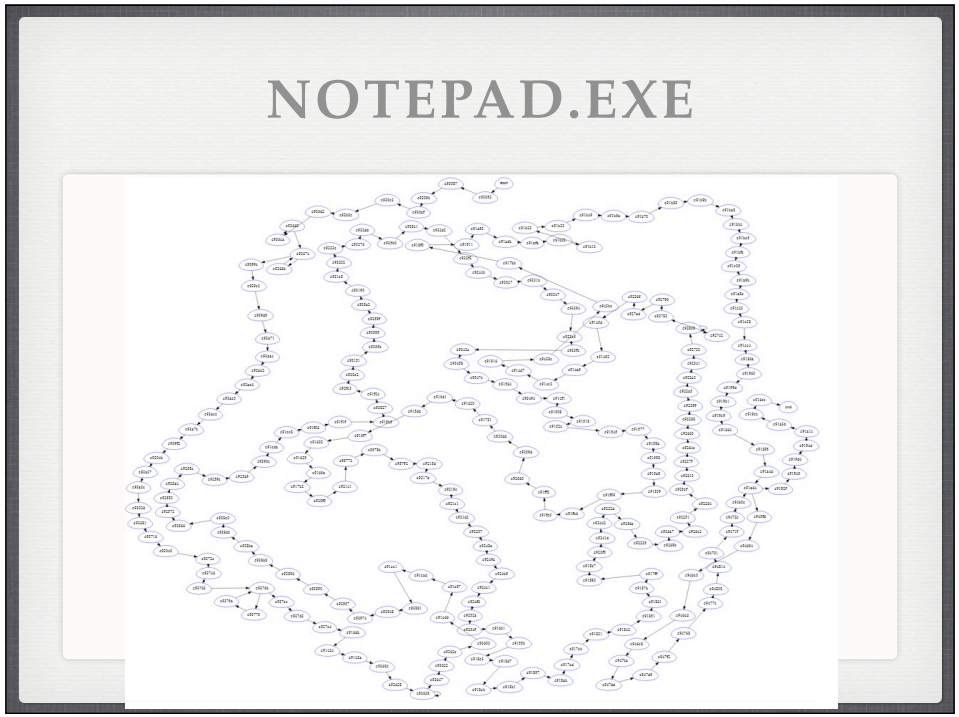
JOHN

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.

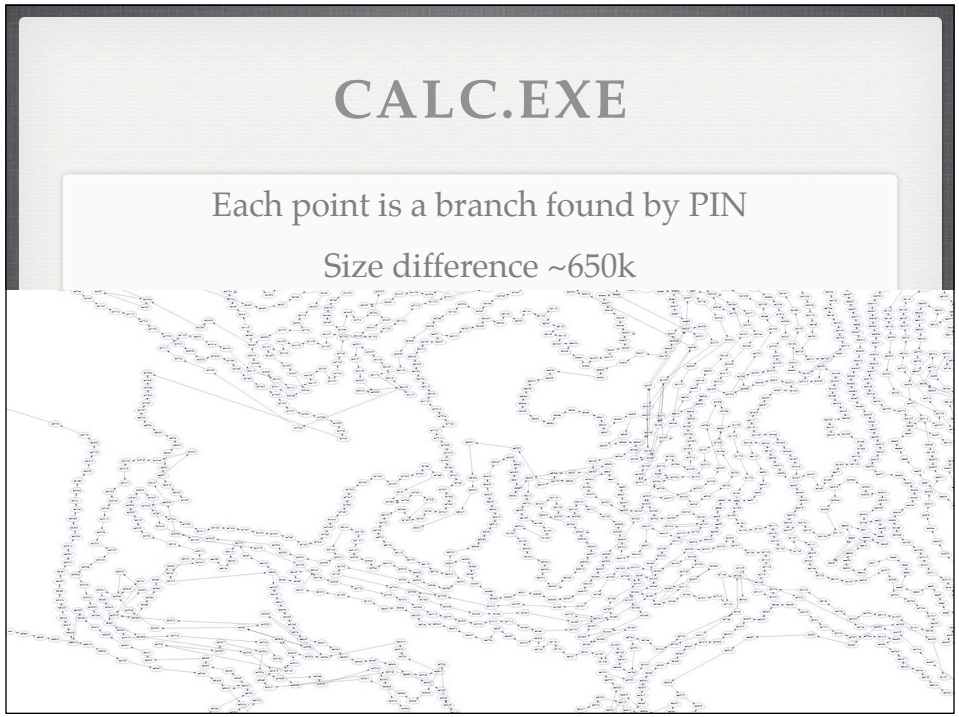
Screenshot is of PIN API used to call a user defined function before each basic block.



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.



JOHN

A bigger amount of instructions increases

CODE COVERAGE: HOW DOES IT WORK?

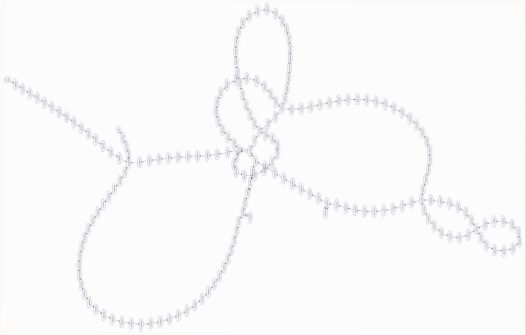
- PIN allows us to base coverage on additional information
  - Stack data
  - register values
  - Caller instead of callee
  - Number of instructions executed before break
  - etc
- Additional flexibility allows for more detailed data

Granularity

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 l
distributed as part of the Pin kit), and the enum definitions are in the file "xed-iclass-enum

Use INS_Mnemonic if you want a string.

Returns:
Opcode of instruction

JOHN

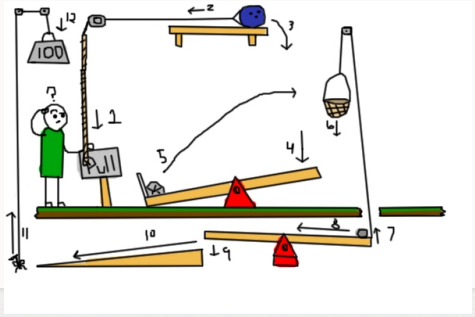
TAINT PROPAGATION

- 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: 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

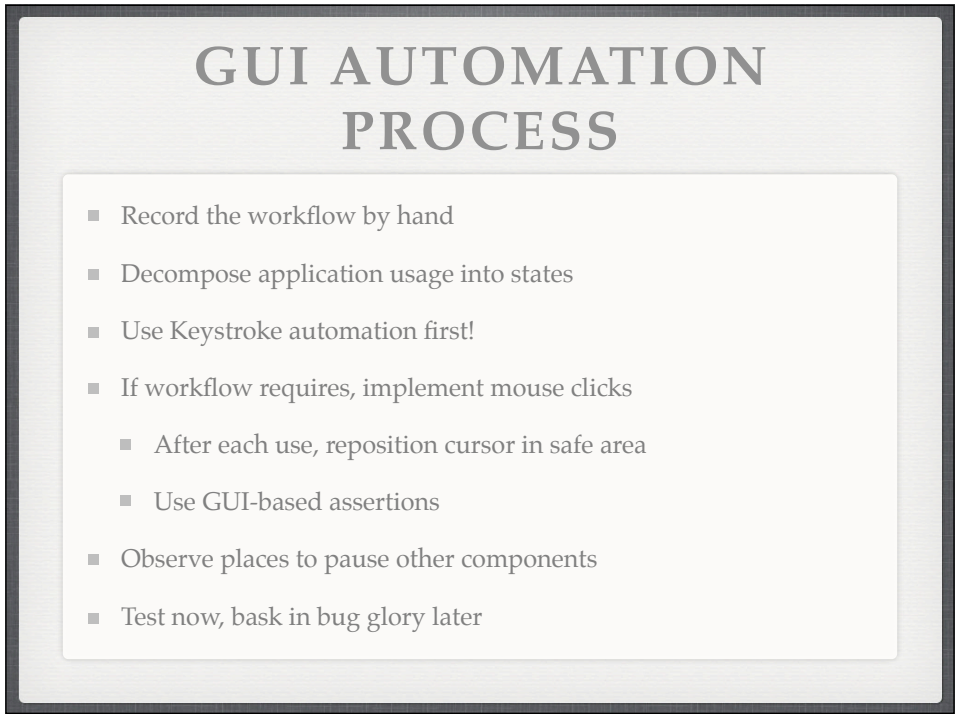
PROTIP: keep directories from which you're accessing things mostly clear, it will reduce dialog population time which can be considerable

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

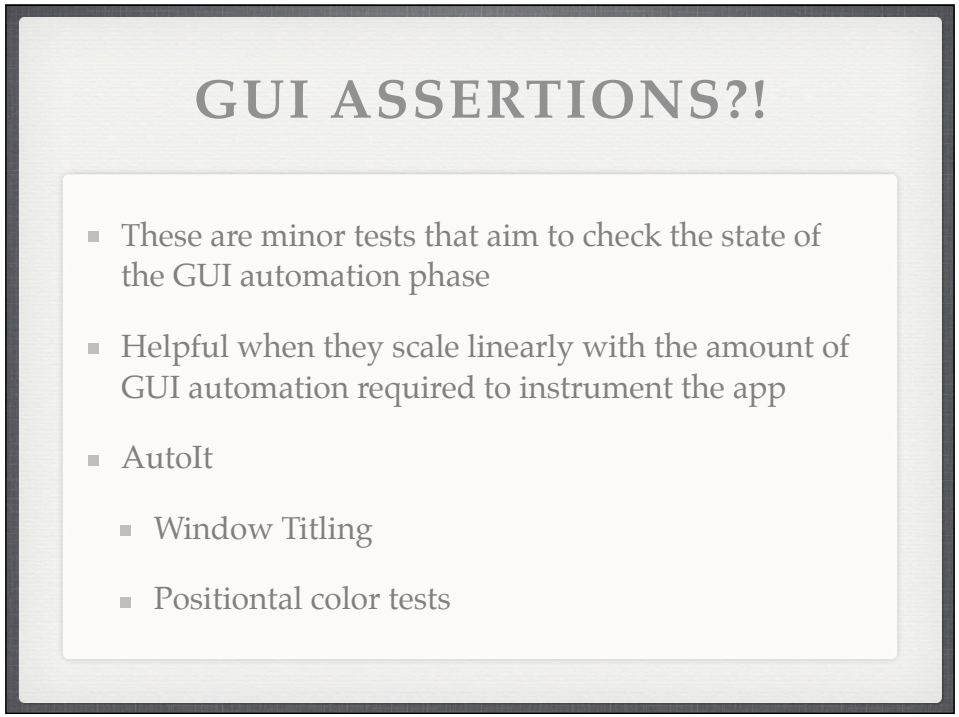
Don't forget calculating pixel positions of things on the screen always will depend on the display resolution. It's almost always better to find the right combination of tab, and arrow keys.

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



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



PETE

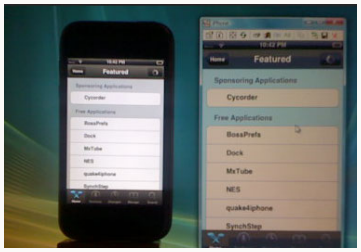
* use these to assert that the GUI automation is in the right state

* 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

Logic vulns can be more subtle, and sneeky 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

```
(f730.ea00): Access violation - code c0000005 (first chance
First chance exceptions are reported before any exception h
This exception may be expected and handled.
eax=00607700 ebx=04f293f8 ecx=ffffa988 edx=0d05a000 esi=00(
eip=676b585a esp=057bfb7c ebp=00005698 iopl=0         nv up
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
*** ERROR: Symbol file could not be found.  Defaulted to ex
Files\VideoLAN\VLC\plugins\audio_filter\liba52tofloat32_plu
liba52tofloat32_plugin!vlc_entry_license__1_2_0l+0x436a:
676b585a 8b0a            mov     ecx,dword ptr [edx]  ds:00
0:012> $<dbgcomm.txt
0:012> !load winext\msec.dll
0:012> !exploitable
*** ERROR: Symbol file could not be found.  Defaulted to ex
Exploitability Classification: UNKNOWN
Recommended Bug Title: Read Access Violation starting at
liba52tofloat32_plugin!vlc_entry_license__1_2_0l+0x000000000000436a (Hash=0x7a532b66.0x7a532b61)
0:012> qq
quit:
```

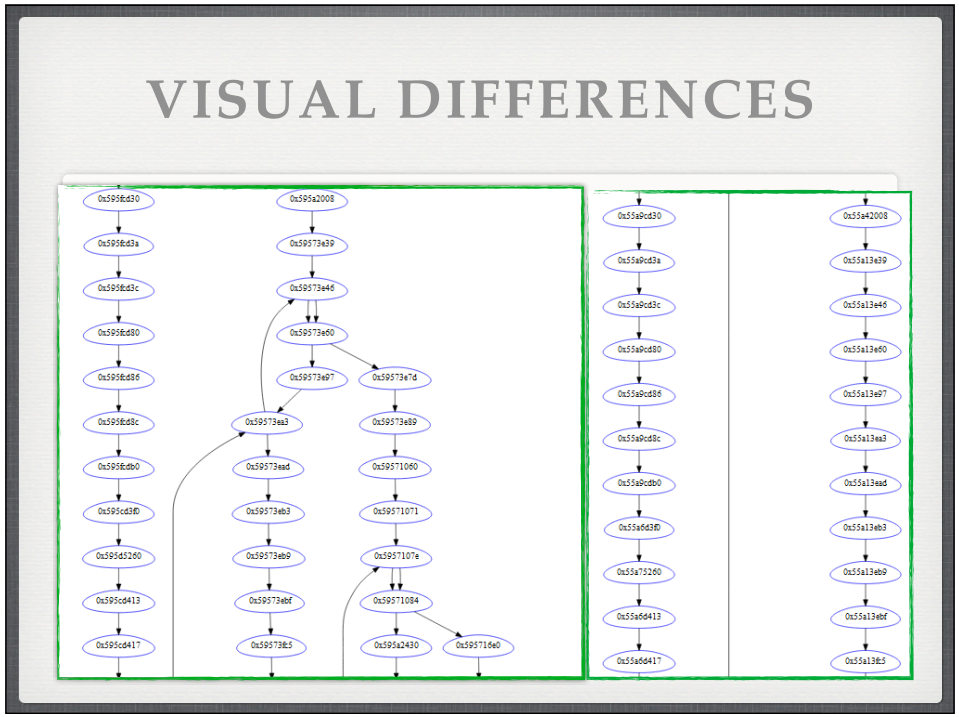
```
(f0a8.ef18): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00400000 ebx=0506ac88 ecx=fffff1a2 edx=0516c000 esi=00000005 edi=00000020
eip=676c585a esp=05a5fb7c ebp=00000e7e iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010216
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\VideoLAN\VLC\plugins\audio_filter\liba52tofloat32_plugin.dll -
liba52tofloat32_plugin!vlc_entry_license__1_2_0l+0x436a:
676c585a 8b0a            mov     ecx,dword ptr [edx]  ds:0023:0516c000=????????
0:012> $<dbgcomm.txt
0:012> !load winext\msec.dll
0:012> !exploitable
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll -
Exploitability Classification: UNKNOWN
Recommended Bug Title: Read Access Violation starting at
liba52tofloat32_plugin!vlc_entry_license__1_2_0l+0x000000000000436a (Hash=0x6453581b.0x64185860)
0:012> qq
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



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 occured.

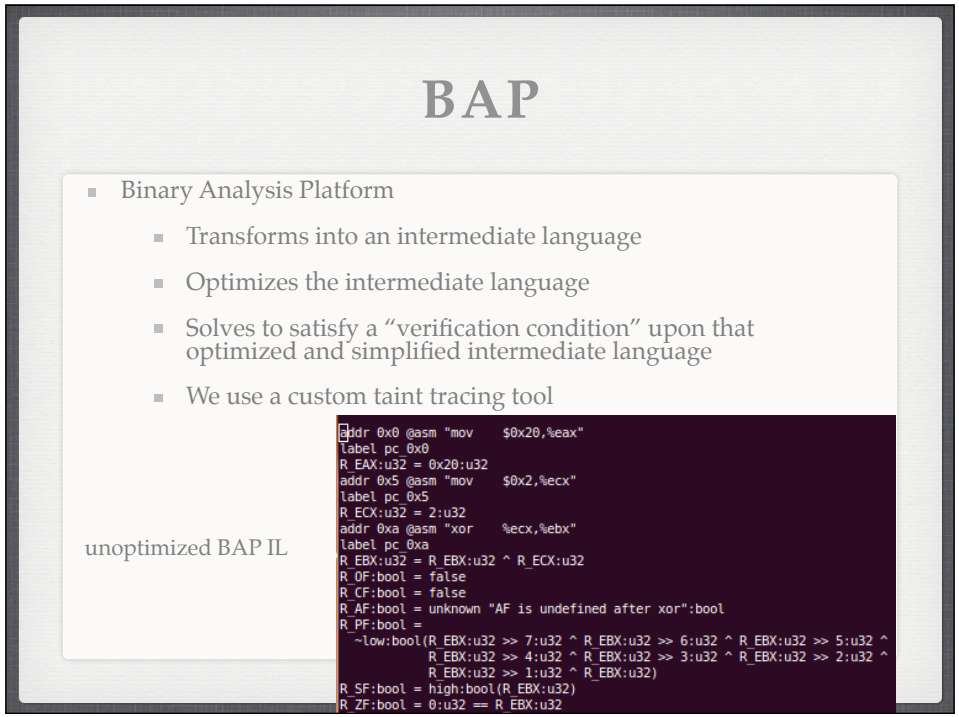
SOLVING FOR CODE COVERAGE

- 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.

```
daybreak@ubuntu:~/Desktop/bap-0.6$ ./utils/topredicate -q -il simp.il -stp-out /tmp/f -post goal -solve
WARNING (Typecheck): infer ast ~check option is deprecated.  Please use typecheck_expression instead.
Hacks: Removing backedges
SSA: Translating to SSA
SSA: Creating new cfg
SSA: Computing defsites
SSA: Computing dominators
SSA: Adding phis
SSA: Added 1 phis
SSA: Grouping phis by block
SSA: Translating block BB_Entry
SSA: translating stmts
SSA: going on to children
SSA: Translating block BB_5
SSA: translating stmts
SSA: going on to children
SSA: Translating block BB_4
SSA: translating stmts
SSA: going on to children
SSA: Translating block BB_Exit
SSA: translating stmts
SSA: going on to children
SSA: Adding 1 phis to the CFG
SSA: Done translating to SSA
Ssa_simp: Starting iteration 0
SSA: rm_phis: removing phi for goal_88:bool
SSA: rm_phis: adding assignments for goal_88:bool
Model:
R_EBX_6_81 -> 0x22
```

BAP is awesome!

JOHN



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