Blind Code Coverage Fuzzing
A part of the Nightmare Fuzzing Suite

Joxean Koret

T2 2014

Who am I?

- My name is Joxean Koret, I'm a Basque Security Researcher working for COSEINC.
  - Previously worked as an independent vulnerability hunter and exploit writer, AV engineer, malware researcher, software developer and DBA.
- I love bug hunting, reverse engineering, writing exploits and software development in no specific order.
- I hate writing ~~auto-fellatio~~ BIO slides but some people recommended me adding 1 and expressed their feelings...

Agenda

- **Introduction**
- Nightmare, the fuzzing suite
- The Blind Code Coverage Fuzzer (BCCF)
- Conclusions

History

- ~2 years ago a friend of mine asked me to do a *fuzzing explained* like talk for an underground conference.
- For this purpose, I started writing a very basic distributed fuzzing suite with web administration.
- To test it, I decided to fuzz AV engines, “somewhat” successfully. See my SyScan'14 presentation for details.
- After a while, this fuzzing testing suite evolved and became one of my main vulnerability hunting tools.
  - As well as my base toolkit for new ideas, like BCCF (Blind Code Coverage Fuzzer).
- This talk is about the fuzzing suite (Nightmare) and the latest research tool I'm writing (BCCF).

Agenda

- Introduction
- **Nightmare, the fuzzing suite**
- The Blind Code Coverage Fuzzer (BCCF)
- Statistics and vulnerabilities
- Conclussions

What is Nightmare?

- Nightmare is an open source distributed fuzzing suite with web administration written in Python.
  - https://github.com/joxeankoret/nightmare
- Nightmare clients can run in any machine and operating system.
  - As long as they can connect to a Beanstalk queue.
- All the server components run in the same server, by default.
  - Database server(s), queue server, mutators, NFP engine and web application could run in different machines if it's ever required.

Architecture (I)

- Database server
  - All information related to crashes (instruction pointer, register values, memory at registers, maps, etc...) is saved in a database (MySQL or SQLite for now).
- NFP engine
  - Responsible of calling mutation engines to generate samples and putting them in a queue.
- Queue server
  - All mutated samples and results, if any, are stored in a queue (Beanstalk) before saving them in the database.
- Web application
  - Web based administration tool for the fuzzing suite.

Architecture (II)

- Clients
  - A client is, basically, a debugging interface for the target operating system, architecture and, sometimes, it's specific to the “system” to be tested.
  - Clients pull off mutated samples from a Beanstalk queue.
  - Clients push back crashing information, if any, in another Beanstalk queue.
  - The NFP engine checks if there is any crashing information in the corresponding queues, pulls off the information from the queue server and stores the data in the database.

Architecture (III)

- Mutation engines
  - Mutation engines are independent tools that are glued to the fuzzing suite using the NFP engine.
  - Mutation engines are external tools called by the NFP engine.
  - A Python script is created to implement mutation algorithms or to interact with external tools (i.e., zzuf or radamsa).
  - Mutation engines typically generate mutated samples + a differences file.
  - Some current mutators: Radamsa & Zzuf wrappers, C. Miller algo, MachO mutator, OLE2 streams mutator, simple replacer, bundles generators, etc...

Architecture, generic clients (IV)

- Generic fuzzers. As of Oct-2014 there are 2 generic fuzzers:
  - generic_fuzzer.py: Fuzzer for applications that receive as command line arguments the file or files to parse/analyse/display.
  - generic_client_server_fuzzer.py: Fuzzer for applications that are client/server. Handles restarting the server process and launching the client with appropriate arguments.
- Future enhancements or additions:
  - GDB, LLDB, WinDBG and IDA wrappers. Currently, it's using VTrace for debugging (and is buggy as hell).

Architecture, analysis tools (V)

- A test-case minimization tool (named generic_minimizer.py) using the generated difference files (if any).
  - At the moment, it can only be used with a limited subset of the mutation engines available.
  - In the future, it will handle any mutation engine even without a differences file.
- Future additions and enhancements:
  - Binary diffing engine to perform mutator agnostic minimization of test-cases.
  - A reproducibility testing tool, exploitability analysers and root cause analysis tools.

Web administration

- Most fuzzing testing suites are built like a set of independent tools.
- Results must be analysed overall by hand.
  - With a few exceptions, like BugMine, created by Ben Nagy.
  - Nightmare, actually, uses many ideas from BugMine.
- They often lack pretty GUI applications to see results, analyse them, downloading repros, etc...
- Nightmare, since day 0, was built with the idea of having an easy to use web GUI.

Web GUI

- Most of the configurable stuff can be done from the web application.
  - With the only exception of the configuration of the web application itself, naturally, and the client fuzzer.
- Projects, mutation engines, the mutation engines associated to each project, etc... are managed from the web application.
- It can be used to find new samples too, if it's required.
- Let's see some pretty pictures of the web application...

Projects management

- Projects can be listed and managed from the web application:

Projects

Action	Name	Description	Subfolder	Tube prefix	Max. samples	Enabled	Archived	Start date
	OpenSSL 1.0.1g	OpenSSL 1.0.1g	ssl	openssl	16	No	No	2014-07-12
	LibreSSL	LibreSSL 2.0 ASN1 decoding code.	ssl	libressl	16	No	No	2014-07-12
	SpiderMonkey	The Mozilla Firefox JavaScript engine, SpiderMonkey. Version 24.2.0.	js	mozjs	16	No	Yes	2014-07-09
	IDA Pro	IDA Pro version 6.6	av	ida	4	No	Yes	2014-07-08
	V8	The Google V8 JavaScript engine	js	v8	16	No	Yes	2014-07-08
	Radare2	Radare2 file formats (/usr/local/bin/rabin2).	av	radare	8	No	Yes	2014-05-23
	[redacted]		[redacted]		8	No	Yes	2014-05-01
	[redacted]		[redacted]		1	No	Yes	2014-03-24
	[redacted]		[redacted]		1	No	Yes	2014-03-21
	[redacted]		[redacted]		1	No	Yes	2014-03-21

26 projec:(s) hidden... Show all projects

Mutation engines

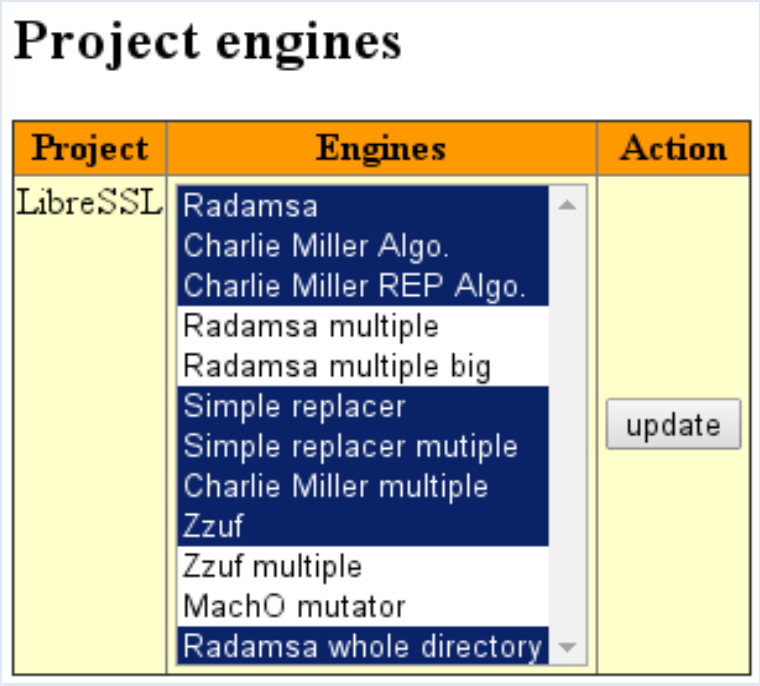
- As well as the mutation engines:

Mutation engines

Action	Name	Description	
	Radamsa whole directory	Use all the samples from the given directory to create mutated samples inferring the grammar.	ra %
	MachO mutator	Half intelligent MachO fuzzer. It only supports a few fields for some headers but, anyway, it should be enough to discover some bugs.	% %
	Zzuf multiple	This is a wrapper using Zzuf mutator to create a zip file with multiple fuzzed samples inside. Useful to fuzz Antivirus engines.	% %
	Zzuf	Zzuf mutator. Preserves input size. http://caca.zoy.org/wiki/zzuf	% %
	Charlie Miller multiple	Same as "Charlie Miller Algo." but fuzzing many files (30) and zipping all of them. It also saves .diff files for the mutated files.	% %
	Simple replacer mutiple	Same as "Simple replacer" but fuzzing many files (30) and zipping all of them. It also saves .diff files for the mutated files.	% %
	Simple replacer	This is a very naïve algorithm: it simply choose a random position in the input file, choose a random character and a random size and replaces a chunk of data with the selected character repeated for the size selected. That's all. It preserves input size and creates a diff file for the mutated file.	% %
	Radamsa multiple big	This is a wrapper using Radamsa to create a zip file with multiple fuzzed samples inside. This is useful, for example, to fuzz Antivirus engines. It creates zip files with 30 samples inside.	% %
	Radamsa multiple	This is a wrapper using Radamsa to create a zip file with multiple fuzzed samples inside. This is useful, for example, to fuzz Antivirus engines.	% %
	Charlie Miller REP Algo.	Similar to Charlie Miller's algorithm but generating repetitions of the same random byte. Preserves input size.	% %
	Charlie Miller Algo.	Python implementation of the random based fuzzing algorithm presented by Charlie Miller in CSW2010. Preserves input size.	% %
	Radamsa	Radamsa is a test case generator for robustness testing, aka a fuzzer. May or may not preserve input size. https://code.google.com/p/ouspg/wiki/Radamsa	ra

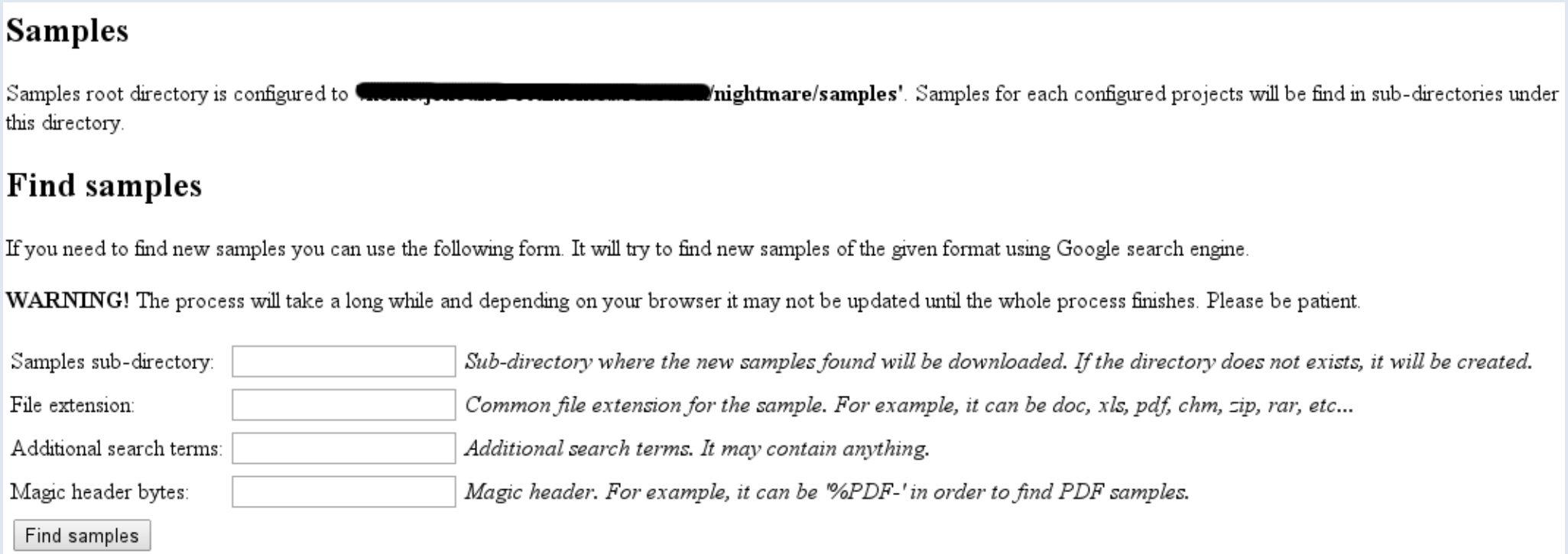
Project's mutation engines

- It can be used, also, to assign mutation engines to projects.
- Not all mutator engines are interesting for all projects, naturally.



Sample files

- We can search new samples (files) using the web application.
  - It simply uses Google for doing so.



Samples

Samples root directory is configured to [redacted]**/nightmare/samples'**. Samples for each configured projects will be find in sub-directories under this directory.

Find samples

If you need to find new samples you can use the following form. It will try to find new samples of the given format using Google search engine.

WARNING! The process will take a long while and depending on your browser it may not be updated until the whole process finishes. Please be patient.

Samples sub-directory:		*Sub-directory where the new samples found will be downloaded. If the directory does not exists, it will be created.*
File extension:		*Common file extension for the sample. For example, it can be doc, xls, pdf, chm, zip, rar, etc...*
Additional search terms:		*Additional search terms. It may contain anything.*
Magic header bytes:		*Magic header. For example, it can be '%PDF-' in order to find PDF samples.*

Find samples

Viewing results

- It can be used to see the results of the projects and download samples, diff files, etc...

Project OpenSSL 1.0.1g - 40 crashes [Download project results]					
Action	Program Counter	Signal	Exploitable	Disassembly	Date
	0x7F[redacted]	SIGSEGV	Unknown	7fa6b9f7cfe9 MOVDQU XMM4, DQWORD [RSI+0x40]	2014-07-14 08:46:27
	0x7F[redacted]	SIGSEGV	Unknown	7f21c0044fe9 MOVDQU XMM4, DQWORD [RSI+0x40]	2014-07-14 08:42:15
	0x7F[redacted]	SIGSEGV	Unknown	7f0481428fe9 MOVDQU XMM4, DQWORD [RSI+0x40]	2014-07-14 07:16:42
	0x7F[redacted]	SIGSEGV	Unknown	7f738f00bfee MOVDQU XMM5, DQWORD [RSI+0x50]	2014-07-14 07:09:12
	0x7F[redacted]	SIGSEGV	Unknown	7fd41b193fe9 MOVDQU XMM4, DQWORD [RSI+0x40]	2014-07-14 05:45:27
	0x7F[redacted]	SIGSEGV	Unknown	7fe196dbefe9 MOVDQU XMM4, DQWORD [RSI+0x40]	2014-07-14 05:29:39
	0x7F[redacted]	SIGSEGV	Unknown	7f903777ffe9 MOVDQU XMM4, DQWORD [RSI+0x40]	2014-07-14 05:02:20
	0x7F[redacted]9D	SIGSEGV	Unknown	None	2014-07-14 03:30:24
	0x7F[redacted]	SIGSEGV	Unknown	7fc76bf83fe9 MOVDQU XMM4, DQWORD [RSI+0x40]	2014-07-14 03:10:07
	0x7F[redacted]	SIGSEGV	Unknown	7f7b1954dfe9 MOVDQU XMM4, DQWORD [RSI+0x40]	2014-07-14 02:37:50
30 crash(es) hidden... Show all crashes.					

Crashing information

- Individual crashes can be inspected in a more detailed view:

Information for crash 3860

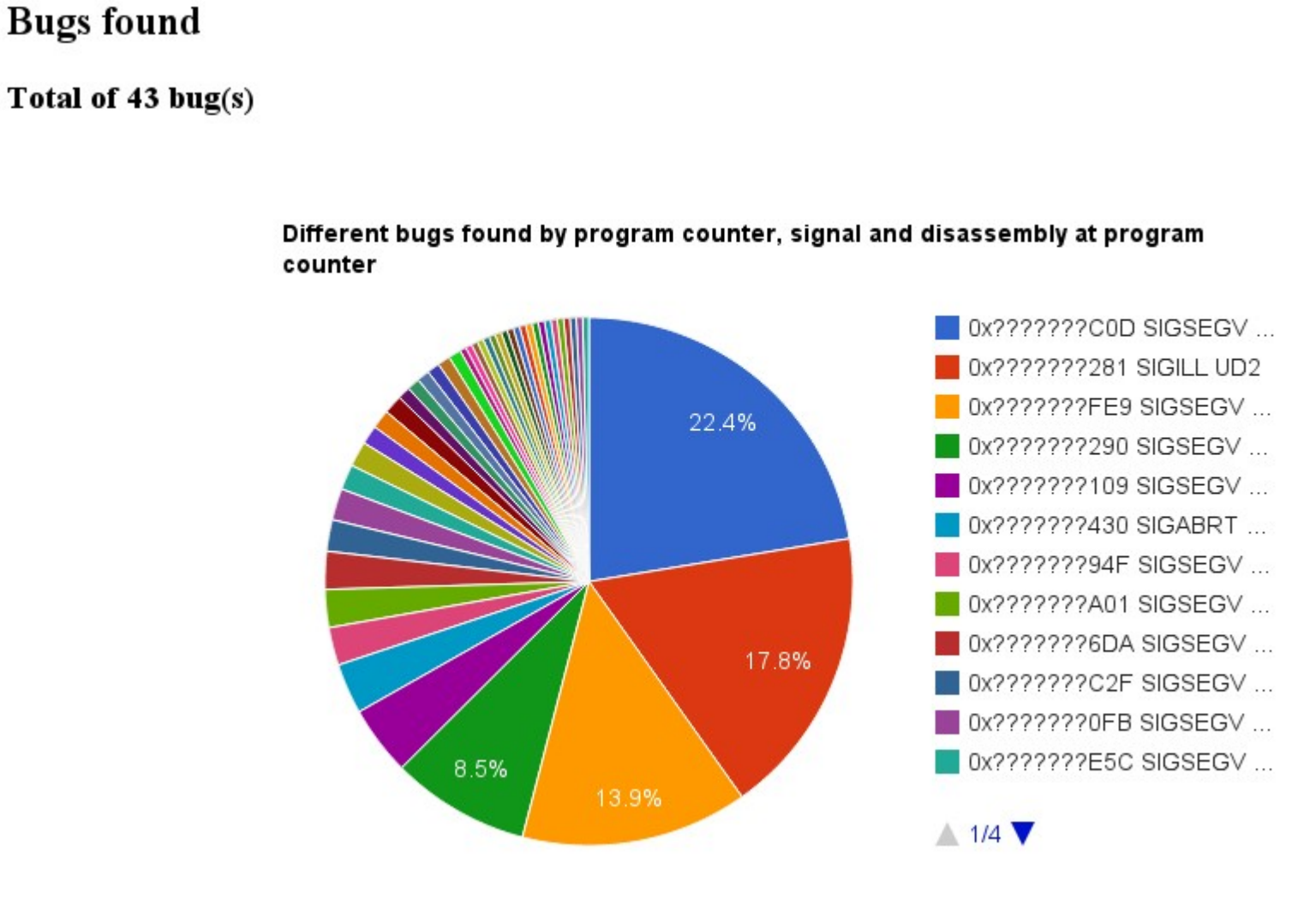
General

Download sample Download diff

Project	IDA Pro
Data	2014-07-08 20:01:20
Signal	SIGSEGV
Exploitable?	Unknown
Program Counter	[redacted]
Crash instruction	f7238fc6 MOVDQU XMM4, DQWORD [EAX+0x40]
Process	PID: 9576
Registers	r14 : 00000000 r15 : 00000000 r12 : 00000000 r13 : 00000000 r10 : 00000000 r11 : 00000000 es : 0000002b r8 : 00000000 rdx : [redacted] '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' rdi : [redacted] '\x88R\x14\n\x01\x00\x00\x00\x01\x00\x00\x00P\r\x17\n\x01\x00\x00\x00\x01\x00\x00\x00 \x1e \x1e\x00\x00\x00\x00' rcx : [redacted] rsi : [redacted] rsp : [redacted] '\xecR\x14\n?<X\xf6\x08\xc0/\xf5"\x13!\n\xa3.@\x00\x01\x00\x00\x00\x00\x00\x00\xff\xff\xff\xff' rbx : [redacted] '|]\x1a\x00\x08[?\xf7\xa0F}\xf7fN\x11\xf7\x10\x83\x17\xf7\x80}\x17\xf7\x90d\x1a\xf7\xa6N\x11\xf7' rbp : [redacted] 'P\r\x17\n\x01\x00\x00\x00\x01\x00\x00\x00 \x1e \x1e\x00\x00\x00\x00!:\\\x13ystem\\Ap' rip : f7238fc6 '\xf3\x0fo`@\xf3\x0fohP\xf3\x0fop`\xf3\x0foxp\x8d\x80\x80\x00\x00\x00\x0f\xae\xe8\x81\xe9\x80' r9 : 00000000 rax : 0a249fba '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' eflags: 00010206
Stack trace	Address: [redacted] / Module:
Disassembly	**0xf7238fc6 MOVDQU XMM4, DQWORD [EAX+0x40]** <---- CRASH

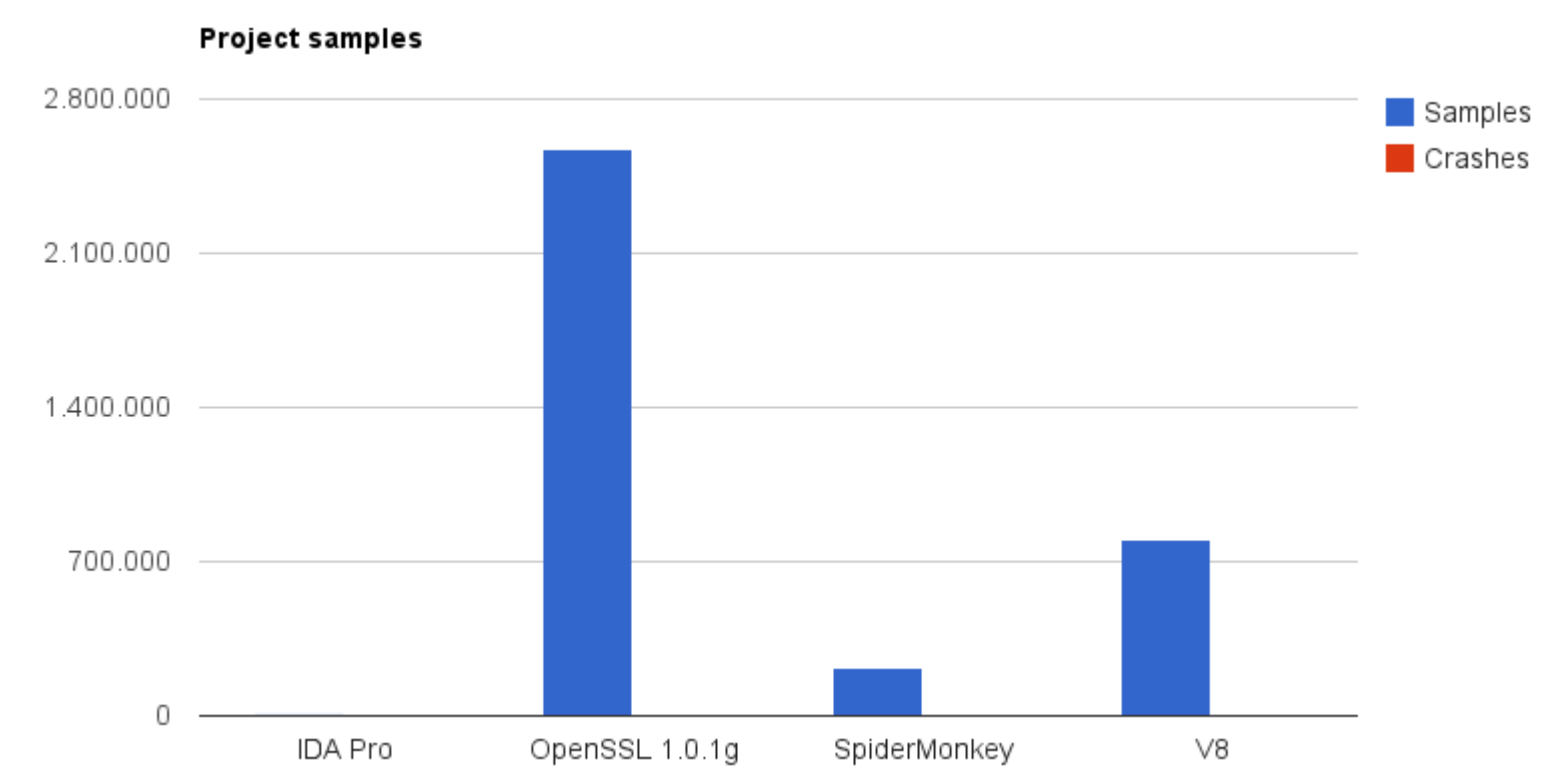
Bugs

- We can see the different number of bugs and the number of times each one appeared:

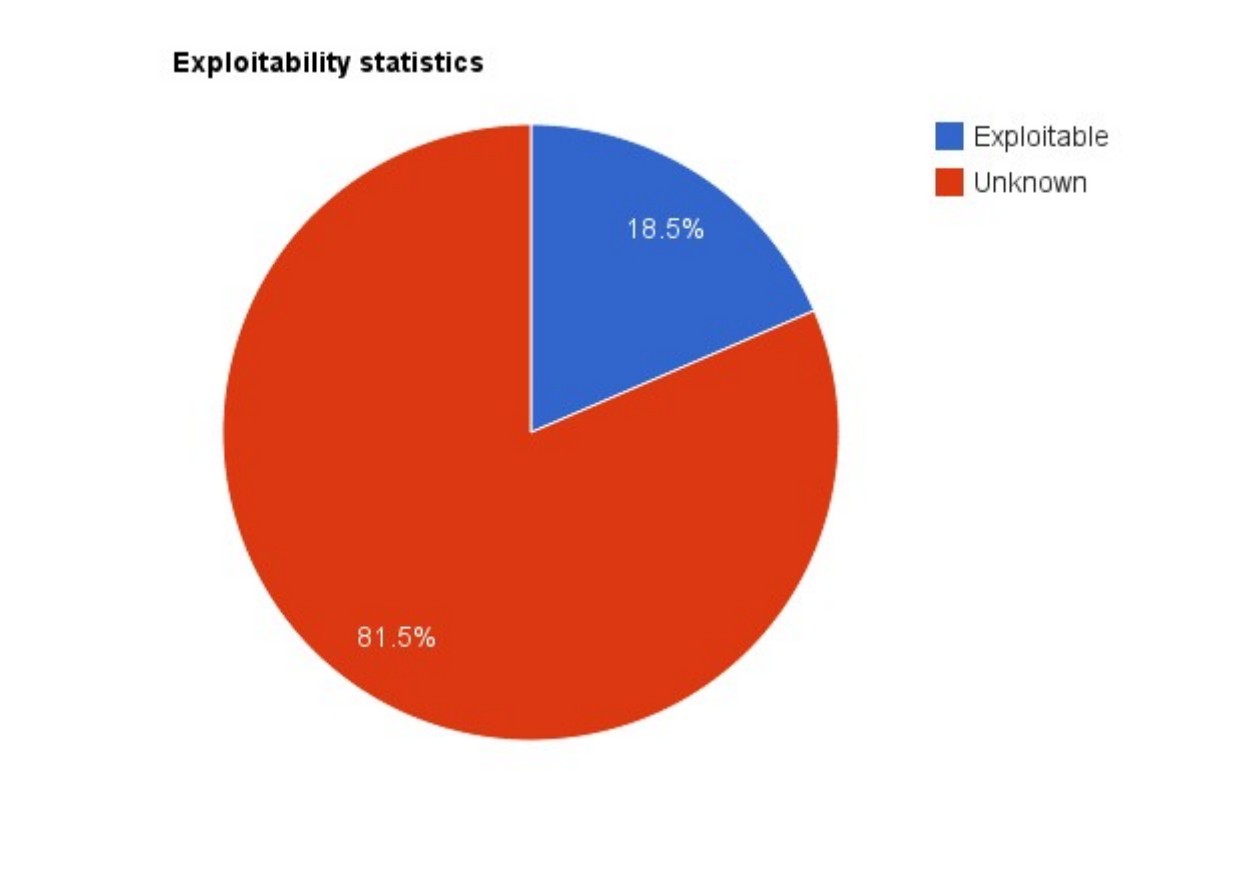


Statistics (I)

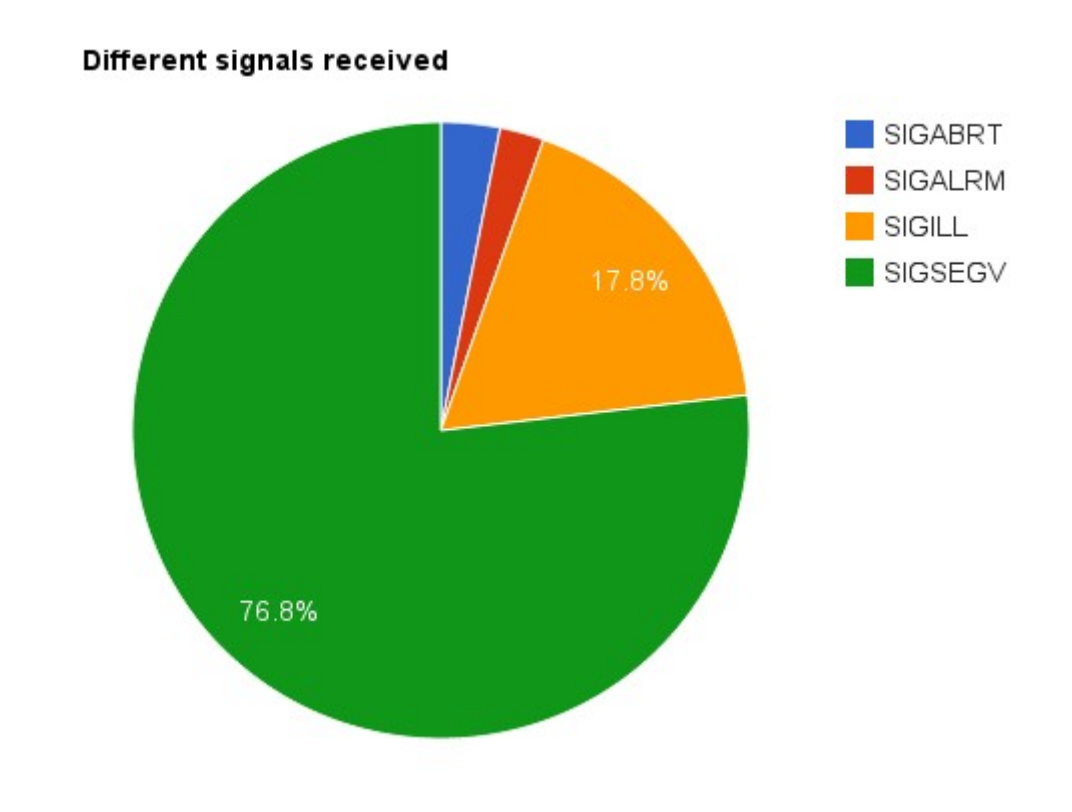
- We can also check the statistics of the fuzzing engine, projects, etc...



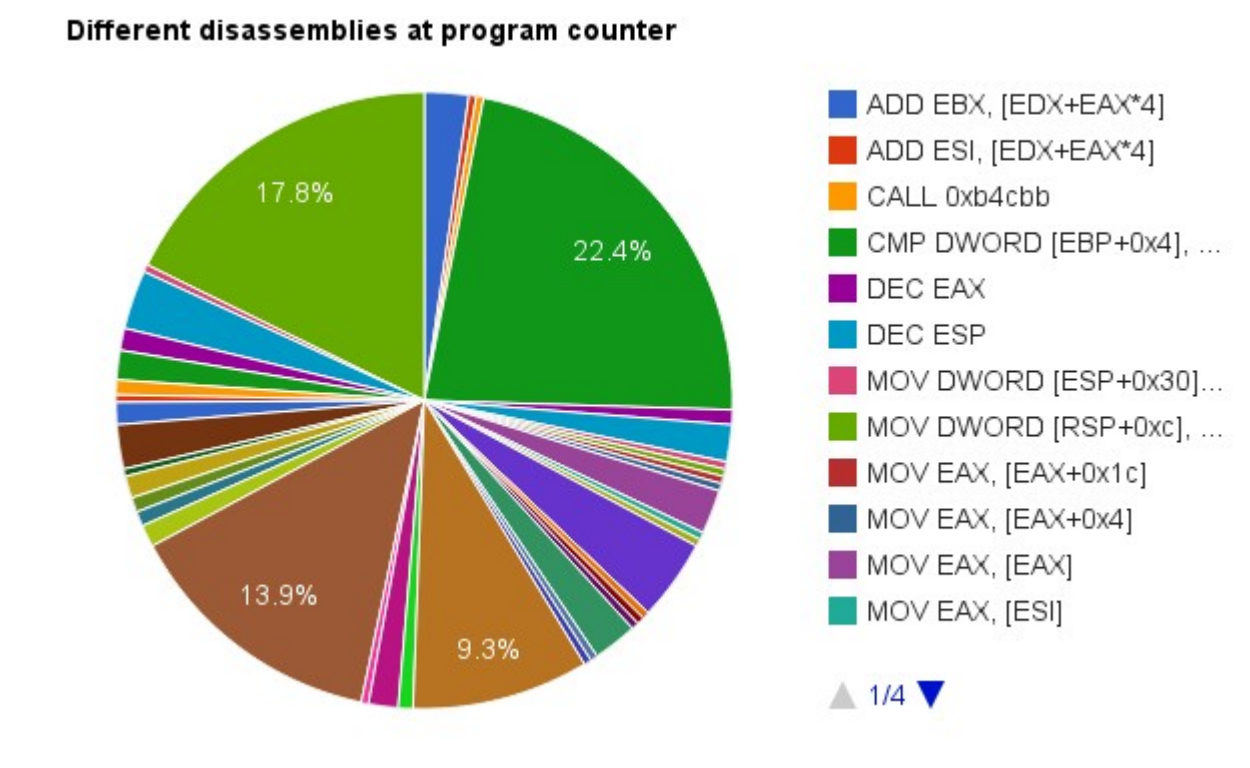
Statistics (II)



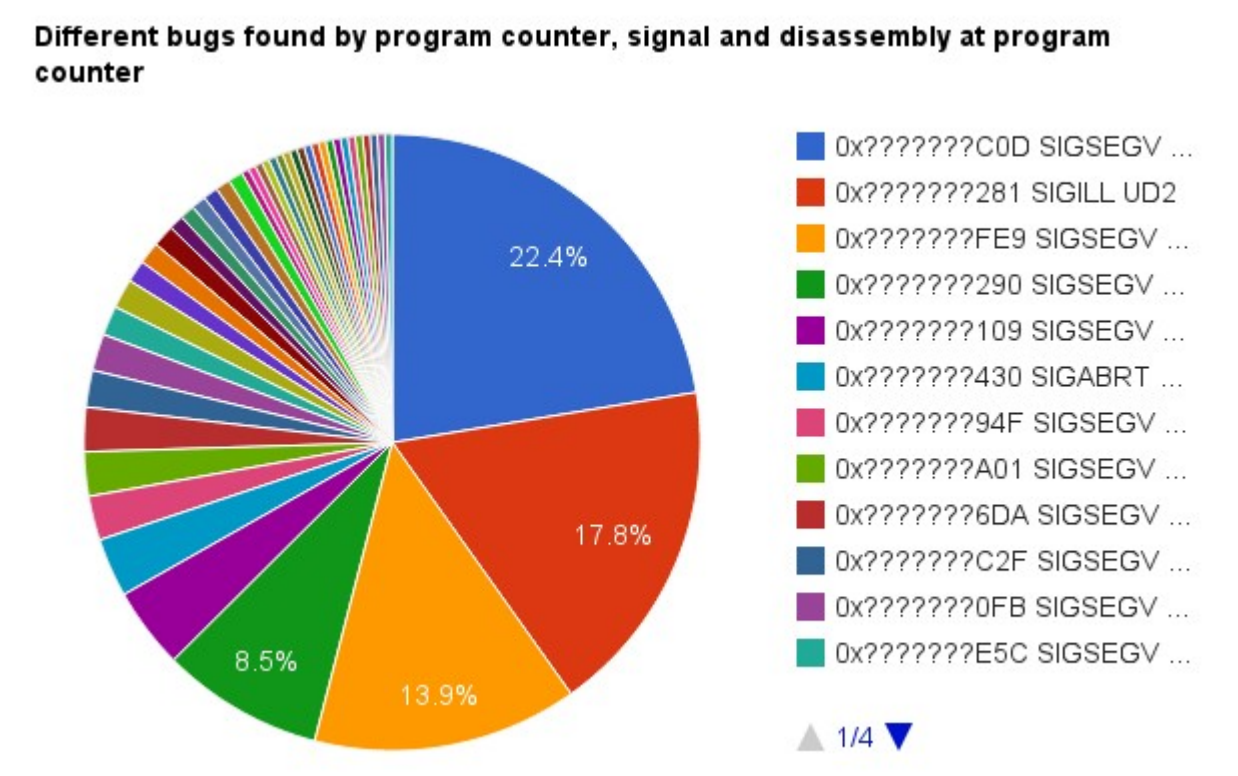
Statistics (III)



Statistics (IV)



Statistics (V)



The Nightmare Fuzzing Suite

- And that's all about the Nightmare fuzzing suite.
- It's open source (GPL v2) so you can download it, use it, modify it (and possibly send patches back to me...) or do whatever you want with it.
  - As long as you accomplish with the GPL v2 license.

Agenda

- Introduction
- Nightmare, the fuzzing suite
- **The Blind Code Coverage Fuzzer (BCCF)**
- Conclusions

Fuzzing and code coverage

- Fuzzing is good at discovering bugs.
  - It's easy to implement (read, creating mutators) and its results, even for the most naïve fuzzers, are incredibly good in many cases.
- However, if we don't check what code we're covering we may be continuously fuzzing the same feature set not discovering bugs hidden in code handling different features.
  - Code coverage is required to determine how many code we're testing.

Code Coverage

- There are various ways we can do code coverage:
  - Using profilers, commonly only useful if we have the source code of the target or full symbols.
  - Using a debugger and tracing the whole application (slow as hell).
  - Or using a binary instrumentation toolkit (still slow, but faster than tracing with a debugger even doing so only at basic block level).
    - DynamoRIO and Intel PIN are the common choices for doing code coverage.
    - In Nightmare, I'm using DynamoRIO, for now.
      - I'll be adding an Intel PIN interface soon.

BCCF, what is this thing?

- BCCF, or Blind Code Coverage Fuzzer, is a tool that tries to perform 3 tasks:
  - Sample files maximization.
  - Generations discovery (new templates to create mutations based on).
  - Bug discovery.
- One specific input (file, for example) will only execute a set of instructions (basic blocks) from the target program.
- If we have a small set of sample files (for using them as templates) we will only exercise the code covered by these samples.

BCCF

- How can we maximize the code exercised by a sample file in a very simple way?
  - Example: we only have a few, a couple or even just one sample for some parser and we want to test as most code/features as possible.
- My idea: using instrumentation, perform pseudo-random modifications and check if more code is executed.
  - Basically, that's all the tool does.
  - However, it's more complex than that...

BCCF, how it works?

- Given an input file and a target application it executes, under instrumentation, the target using the input file a number of times (10 by default) to extract some statistics.
  - Minimum, maximum and average number of **different** basic blocks.
- After the initial statistics are gathered, it will perform modifications to the input file and launch again the target under instrumentation using the modified input file.
- ...

BCCF, how it works?

- If more different basic blocks are executed with the applied modifications, a new generation is created and saved in memory, as well as the corresponding statistics for this generation.
  - I say I use “blind code coverage” as I'm not checking which basic blocks are being executed, only if the number of basic blocks is higher.
- This new generation (the modified input) is used as the template for applying new almost random modifications.
- If new basic blocks are executed, again, a new generation is created, preserving a maximum of X generations (10 by default) in memory.
- ...

BCCF, how it works?

- Modifications applied to the input buffer are generated using:
  - An iterative algorithm (i.e., changing each byte from the input to each possible value, 00 to FF). Very slow, naturally.
  - Or a random modification of a byte, word, dword or qword. Ultra-fast but often gives bad results.
  - Or using Radamsa. Fast and gives very good results as it tries to infer the grammar and generate inputs accordingly.
  - Or using, for each iteration, a random selection of any of the previous algorithms.
  - And, soon, using any mutator from Nightmare, thus, possibly using format aware mutators for which results should be very good.
- ...

BCCF, how it works?

- The current generation (the last modified buffer) is being used for applying new changes to it using any of the previously explained methods.
- If after a number of new changes applied to the generation it either lowers the number of different basic blocks executed or it remains stable, the current generation is dropped and the previous discovered one, if any, is pulled off from the list of generations.
- ...

BCCF, how it works?

- The tool can run continuously, forever, or can be run for a number of iterations until the given input file is maximized.
- If it runs forever, it can be used to discover vulnerabilities and the results will be stored, for now, on disk.
  - Naturally, tight integration with all the Nightmare fuzzing suite is planned.
- If it's instructed to run for a number of iterations, it can be used to just maximize a number of files for using them later on as template files.
  - But some bugs in ~~stupid~~ targets will be discovered.

BCCF, how it works?

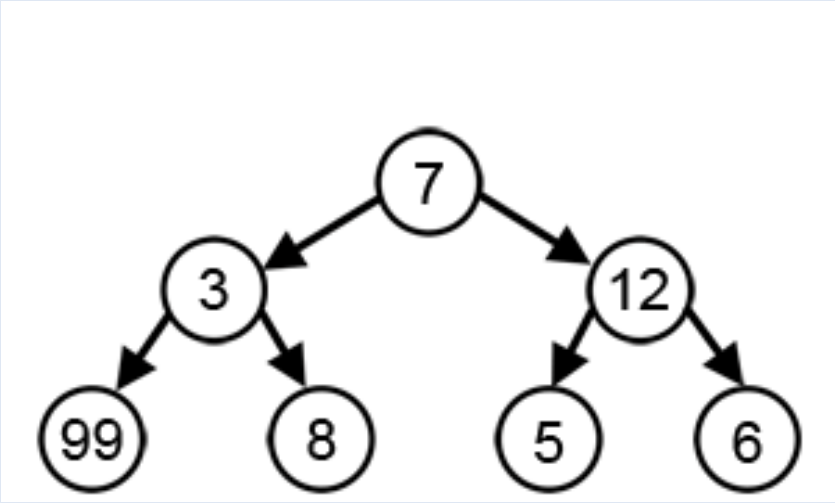
- When running continuously it can also be used to create generations for using them as new templates.
  - BCCF generates templates in a template's directory.
  - NFP engine runs mutators against the new generations (with more features than the original template file).
  - The results of using BCCF in this way are often very good.

BCCF: problems

- BCCF uses a greedy algorithm to discover as most code as possible.
- However, it tries to recover from errors dropping generations considered bad.
- It does this for a good reason.
- Let's see the next slide...

BCCF problems: Greedy algorithms

- Consider each node's value the number of basic blocks.
- A not so intelligent greedy algorithm trying to discover more code would discover only the path 7-12-6, as it's the one executing more code when not recovering from errors.
- A greedy algorithm recovering from errors, however, would likely discover the true longest path, 7-3-99.



- Also, given enough time, it could discover all paths.

BCCF problems: Greedy algorithms

- BCCF does, I think, a good work in recovering from errors as explained in previous slides.
- However, it only means it's going to execute more code.
- It, unfortunately, doesn't mean is going to discover always more bugs.
- Why? Because executing more different basic blocks doesn't mean we're going to find a bug in that code.
  - Statistically, in theory, the odds are higher. But, depending on the target, the practice can be different.
- Also, it's not going to discover changes that depends on, say, checksums, cryptographic hashes, etc... naturally.
  - Until support for format aware mutators is added...

Uses of tools like BCCF

- One interesting property of a tool like BCCF is that, with a small change, it can be used to discover infinite loops and memory exhaustion bugs.
  - Perhaps not the most interesting bugs, yeah...
  - But, nevertheless, an interesting property I think.
- We only need to check the number of basic blocks executed without checking the number of different ones.
  - Thanks to this property, I discovered various remote infinite loop bugs in AV engines.

Uses of tools like BCCF

- As we can use this tool to discover infinite loops and memory exhaustion bugs, it can be used to actually determine which fields/offsets in a given input file are used for:
  - Loop conditions. In other words, we can blindly discover the offsets used to read the size of various elements in a given input file.
  - Memory allocations. We can blindly discover the offsets used to read the size of memory blocks that will be allocated.
  - I plan to add heuristic mutations for fields/offsets when allocations or loop conditions are discovered.
    - For example: Try size 0, -1, etc...

Uses of tools like BCCF

- Another interesting property of a tool like BCCF:
  - It can, given enough time, discover many properties of the file format, blindly.
- It isn't as smart as SAGE or similar tools, like the one demoed by Edgar Barbosa in Syscan360'14 or the SAGE clone presented by the VRT team at Shakacon & RECON'14, that uses SAT/SMT solvers to determine the new inputs that must be created in order to exercise more code.
  - However, it's far easier to implement and requires notably less resources.
- ...

Uses of tools like BCCF

- An easy example:
  - Create a file with the magic header of the file format you want to test and fill it with 0x00 characters up to the size you want.
  - Run BCCF using this dumb file as the template.
  - Given enough time, it will discover many modifications that makes the target to execute more code, thus, discovering more details about the file format/protocol.
  - At the same time, it will discover also bugs.

DEMO
readelf

Example execution of BCCF

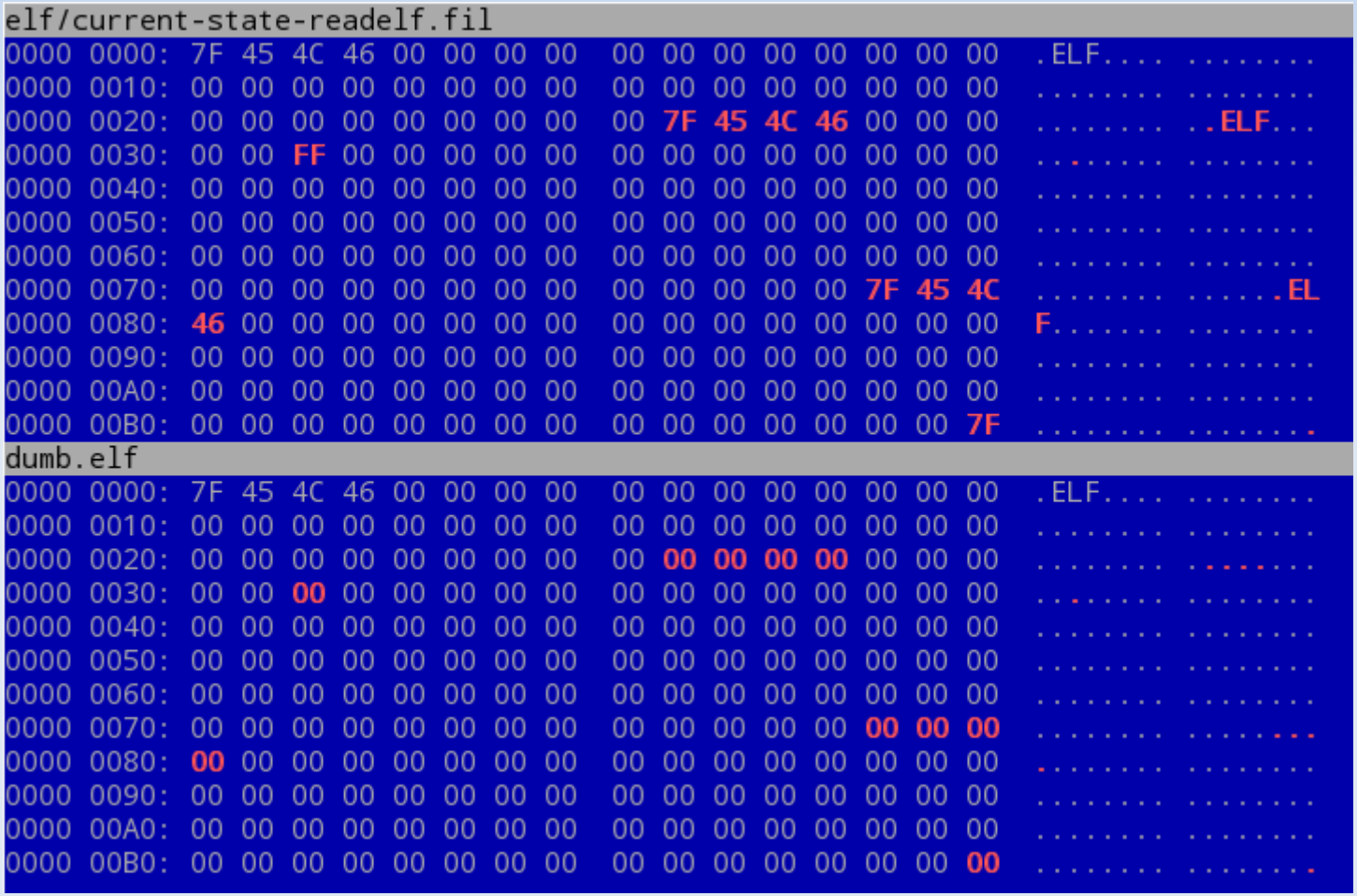
- Using as target readelf and a dumb ELF file (as explained in a previous slide):

```
[Mon Jul 21 13:25:34 2014 19821:140706589271808] Selected a maximum size of 4 change(s) to apply
[Mon Jul 21 13:25:34 2014 19821:140706589271808] Input file is dumb.elf
[Mon Jul 21 13:25:34 2014 19821:140706589271808] Recording a total of 10 value(s) of coverage...
[Mon Jul 21 13:25:35 2014 19821:140706589271808] Statistics: Min 1681, Max 1681, Avg 1681.000000, Bugs 0
[Mon Jul 21 13:25:35 2014 19821:140706589271808] Fuzzing...
[Mon Jul 21 13:25:39 2014 19821:140706589271808] Iteration 100, current generation value -300, total generation(s) preserved 0
[Mon Jul 21 13:25:44 2014 19821:140706589271808] Iteration 200, current generation value -600, total generation(s) preserved 0
[Mon Jul 21 13:25:49 2014 19821:140706589271808] Iteration 300, current generation value -900, total generation(s) preserved 0
[Mon Jul 21 13:25:53 2014 19821:140706589271808] Iteration 400, current generation value -1200, total generation(s) preserved 0
[Mon Jul 21 13:25:53 2014 19821:140706589271808] GOOD! Found an interesting change! Covered basic blocks 1764, original maximum 1681
[Mon Jul 21 13:25:53 2014 19821:140706589271808] New statistics: Min 1681, Max 1764, Avg 1722.500000
[Mon Jul 21 13:25:58 2014 19821:140706589271808] Iteration 500, current generation value -102, total generation(s) preserved 1
[Mon Jul 21 13:26:01 2014 19821:140706589271808] GOOD! Found an interesting change! Covered basic blocks 1773, original maximum 1764
[Mon Jul 21 13:26:01 2014 19821:140706589271808] New statistics: Min 1764, Max 1773, Avg 1768.500000
[Mon Jul 21 13:26:04 2014 19821:140706589271808] Iteration 600, current generation value -45, total generation(s) preserved 2
[Mon Jul 21 13:26:09 2014 19821:140706589271808] Iteration 700, current generation value -153, total generation(s) preserved 2
[Mon Jul 21 13:26:09 2014 19821:140706589271808] GOOD! Found an interesting change! Covered basic blocks 1797, original maximum 1773
[Mon Jul 21 13:26:09 2014 19821:140706589271808] New statistics: Min 1773, Max 1797, Avg 1785.000000
```

- As we can see, it discovered various modifications that caused the target to execute more code.
- We can diff the current generation with the template file to see the modifications.

Example modifications

$ vbindiff dumb.elf elf/current-state-readelf.fil



Example output of the target

```
joxean[redacted]/nightmare/fuzzers$ readelf -a elf/current-state-readelf.fil
ELF Header:
  Magic:   7f 45 4c 46 00 00 00 00 00 00 00 00 00 00 00 00
  Class:                             none
  Data:                              none
  Version:                           0
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              NONE (None)
  Machine:                           None
  Version:                           0x0
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               0 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         32512
  Size of section headers:           19525 (bytes)
  Number of section headers:         70
  Section header string table index: 0
readelf: Error: Unable to read in 0x14dade bytes of section headers
readelf: Error: Section headers are not available!
```

And after a while...

```
[Mon Jul 21 13:43:57 2014 19821:140706589271808] Dropping current generation and statistics as we have too many bad results
[Mon Jul 21 13:43:57 2014 19821:140706589271808] Statistics: Min 2368, Max 2369, Avg 2368.500000, Bugs 1
[Mon Jul 21 13:43:57 2014 19821:140706589271808] Iteration 19580, current generation value -266, total generation(s) preserved 9
[Mon Jul 21 13:43:58 2014 19821:140706589271808] Iteration 19600, current generation value -306, total generation(s) preserved 9
/bin/sh: línea 1:  3837 Violación de segmento  (`core' generado) [redacted]DynamoRIO-Linux-4.2.0-3//bin64/drrun
n/readelf -a "/tmp/tmp7WWwht.fil" > /dev/null 2> /dev/null
[Mon Jul 21 13:44:04 2014 19821:140706589271808] *** Found a BUG, caught signal 139 (SIGSEGV), hurra!
[Mon Jul 21 13:44:04 2014 19821:140706589271808] Output path configured to elf/
[Mon Jul 21 13:44:04 2014 19821:140706589271808] Created proof of concept elf/53df49abf34a10dee66babc2f11592875cd3d146
[Mon Jul 21 13:44:04 2014 19821:140706589271808] Created diff file elf/53df49abf34a10dee66babc2f11592875cd3d146.diff
```

- I selected this target to make a demo because it's incredibly easy to make it to crash.
  - It's plagued of bugs and also uses abort() in so many places.
- If you used to trust this tool, you should not...
- I use this tool to test my fuzzers...
  - In the past, I used xpdf...

More use-cases of tools like BCCF

- One more usage: Let's say that you only have one or 2 samples for some rare file format.
  - Or some likely problematic/complex sample, or a sample that already caused problems and is likely going to cause more...
- You can run BCCF for each sample and discover new functionality (generations).
  - New generation files will be written out to disk.
- You can then use these new generations as templates to mutate from.
  - As new generations will cover new functionality not covered by the original templates.
- Just one more idea. And probably the best usage scenario for BCCF, I think.

DEMO
BitDefender

Problems of tools like BCCF

- One of the obvious problems of fuzzing with a tool like BCCF: the required time.
  - How long would it take for such a tool to discover the crash in the next example?

```
void test(char *buf)
{
    int n=0;
    if(buf[0] == 'b') n++;
    if(buf[1] == 'a') n++;
    if(buf[2] == 'd') n++;
    if(buf[3] == '!') n++;
    if(n==4) {
        crash();
    }
}
```

DEMO
bad_test

Problems of tools like BCCF

- OK, the previous one is doable in very little time... What about this one?

```
void test(char *buf)
{
  char *myhash = "654e415ca2c9cde0e41682b6df163829db79454dc60ebbd4326ad4059f806b30";
  if ( check_sha256(buf, myhash) )
    crash();
}
```

- Either impossible or requires a format aware mutator (which is not exactly *blind*).
  - Although a format aware mutator can be used with BCCF, why not, and, indeed, is going to be added soon. It would still be blind as it doesn't know the inputs that must be created, it would only know how to fix some fields.

Problems of tools like BCCF

- Some of these problems can be “fixed” by using mutators aware of the format.
- But such a tool cannot be called exactly *blind*...
  - Although, I remark, the changes would be blind.
  - The format aware mutator would fix the corresponding fields to consider the file format “not malformed”.

Similar tools

- The most similar tool to BCCF I know is AFL, American Fuzzy Lop, by Michal Zalewsky.
- AFL works with source codes and it instruments every line of source code recording a tuple with
  - [ID of current code location], [ID of previously-executed code location]
- If new tuples are discovered, a good change was discovered and a generation is created and put in a queue.
- ...

Similar tools

- However, this approach have some problems, in my opinion:
  - Source code is required, it doesn't work with binaries as is distributed.
  - It could be adapted to a PIN/DynamoRIO tool easily, I think.
  - It doesn't use format specific or intelligent mutators (radamsa...) but “traditional fuzzing strategies”.
- But it also have some good advantages:
  - It's using pure source code instrumentation, thus, it's faster than common binary instrumentation.
  - It doesn't mess with libraries code.

Similar tools

- The other most similar tool is from another Googler: Flayer, by Tavis Ormandy.
  - The tool is similar but way more intelligent than mine.
  - They call their approach “Feedback driven fuzzing”.
  - Flayer uses DBI: it's a Valgrind tool and does sub-instruction profiling instead of basic block profiling.
- The only drawbacks I can see from this tool:
  - Valgrind is sssssllllllllooooooowwwww as hell.
  - And is not mine ;) I typically prefer to write my own tools, over all.

Other similar tools

- There are many other tools with a similar idea: assisting fuzzing with code coverage.
  - AutoDafe. Uses a technique called “fuzzing by weighting attacks with markers”.
    - Prioritizes test cases based on which inputs have reached dangerous API functions.
  - EFS, by Jared DeMott. Evolutionary Fuzzing.
    - Uses genetic algorithms to determine the inputs.
- EFS is may favourite from this list.
  - However, its complexity is very high: too many parameters.

And more “similar” tools

- Microsoft SAGE is another similar tool. However, their approach is completely different. A very brief explanation of it:
  - SAGE performs code coverage and taint analysis of inputs and, then, converts the executed code to SMT formulas.
  - The SMT solver Z3 is feed with them and it outputs possible changes to be applied to the input data in order to maximize code execution.
  - According to them, 1/3 of the bugs in applications like Microsoft Word are discovered with this approach.
  - The most elegant approach.
- Drawbacks:
  - Don't expect to run it at home against real targets today.
  - The complexity of developing such an application is, not surprisingly, extremely high.
  - Elegant doesn't mean “best approach”.

Agenda

- Introduction
- Nightmare, the fuzzing suite
- The Blind Code Coverage Fuzzer (BCCF)
- **Conclusions**

Conclusions

- Fuzzing is still one of the best automated methods to find vulnerabilities.
  - Even rudimentary initial proof-of-concepts of new ideas actually find bugs.
- Contrary to the believe of many people, there are still many areas where fuzzing can, and will, be improved.
- Code coverage greatly helps in assisting fuzzing.
  - Intelligent fuzzing using code coverage is, in my opinion, the (present and) future of fuzzing.

Thanks for your time!
Questions?