

Tavis Ormandy Google, Inc.

Making Software Dumber

Tavis Ormandy Google, Inc.

Making Software Dumber

- This talk will discuss some of the ideas we've explored while thinking about trying to make more generic fuzzers.
- While a lot of fuzz related research has focussed on making fuzzing tools more aware of the protocol their attacking, some of us have felt that this may be the wrong direction
 - We want to make very generic fuzz testing tools that can apply to lots of software.
 - We've been calling this "Making software dumber", as opposed to making fuzzers smarter.



Fuzz Testing

- In it's purest form, entirely blind to context and underlying protocol.
- Historically, this approach has proven to be remarkably successful.
- However, it is self-evident that this approach remains limited to software expecting only minimally structured input.
- Applying the core principles of fuzz testing to a broader range of software and improving it's overall efficacy continues to be an active research topic.



Block Based Fuzzing

- Perhaps the earliest attempt at introducing structure to fuzz testing.
 - Pre-define the data structures (i.e. blocks) involved in the protocol being tested.
 - The blocks are then assembled and mutated in such a way that the basic structure is maintained, while the contents and stream are randomly modified.
- The most notable example, of course, was SPIKE.
- SPIKE et al. began to vastly extend the reach of fuzz testing to structured protocols and formats (HTTP, RPCs, XML, etc).



Model Inference Assisted Fuzzing

- An extension to block based fuzzing, Model Inference Assisted
 Fuzzing Introduces protocol awareness in order to extend the reach of
 fuzz testing to vastly more complex programs
 - Predefined protocol grammar serves as a complete specification for the protocol (or format) to be tested.
 - The fuzzer is then able to generate tests that deviate subtly from those specifications
- Model inference assisted fuzzers continue to expose serious implementation flaws.
- Perhaps the most notable example is PROTOS.



Model Inference Assisted Fuzzing

- Model inference assisted fuzzing is obviously a considerable leap forward from naïve protocol-blind fuzzing.
- However, reliance on accurate protocol specifications presents a number of problems
 - Expensive setup cost required to construct the requisite grammar.
 Some specifications provide usable grammar in Backus-Naur or similar form, but this is an exception rather than a rule.
 - Only possible to model specifications documented by the vendor, potentially ignoring any undocumented or proprietary extensions.
 - High likelihood of ignoring vast amounts of attack surface, or testing large amounts of unimplemented specifications (consider testing a http daemon that doesn't support DAV based purely on the specification).



Possible Solution: Feedback Driven Fuzzing?

- Feedback driven fuzzing attempts to learn how to explore a program dynamically (typically) using code coverage and sample inputs.
 - Generally either use compiler instrumentation (such as -finstrument-functions in gcc) or DBI.
 - The three major DBI frameworks are PIN from Intel, DynamoRIO from Determina/VMware, and VEX/Valgrind from Julian Seward et al, But others have used home-brew techniques (generally IDA +patching basic block boundaries with software breakpoints).
- Bunny-the-fuzzer (Icamtuf) and EFS (DeMott) are two notable examples.
- Feedback driven fuzzing has proven to be effective, and continues to be an exciting research area.



Alternative Inference Sources

- We're convinced model inference assisted fuzzing is useful, but we wanted comparative results without the expensive (in terms of effort) initial investment.
- We developed an alternative solution that can be almost entirely automated with minimal human interaction.
 - We have been able to apply this to explore proprietary, and undocumented software functionality
 - Identify edge cases that require special attention, and automatically generate (surprisingly) good quality regression test suites.
 - We successfully used this method to find multiple real life security problems in a large number of unrelated products.



Feedback Driven Fuzzing

- So, Inspired by ideas such as Protos, EFS, Bunny, SAGE, and others, We wanted to build code coverage feedback into our fuzzing.
- Initially we simply hooked gcc's built in gcov instrumentation support
 which modifies every basic block to increment a counter we can
 monitor. It's not a supported interface, but suited our needs well.
- This worked, but requires us to have the source for every application we want to test.
- Using DBI, we can apply the same logic to any application but do not require the source code.
- Additionally as the major DBI frameworks are cross platform, we get Windows support for free.



- Inspired by model inference assisted fuzzing, corpus distillation eliminates the requirement for protocol grammar via automated observation of the software to be tested.
- We realised that the input to any program could be considered a set of elements from the finite universe of source code lines (or basic execution blocks) that form the program you're testing.
- In this manner, the input X is the subset of lines from program P that have been executed one or more times for P(X).
- SAGE From Microsoft Research had similar goals, but a vastly different approach (and more complete, but we address this later).



```
bool decode(FILE *infile)
11
12
        png_structp png_ptr;
13
        png_infop info_ptr;
14
        png_ptr = png_create_read
15
        info ptr = png create inf
16
        png set crc action(png pt
        if (setjmp(png_jmpbuf(png
17
18
            png_destroy_read_stru
19
            fclose(infile);
20
            return false;
21
22
        png_init_io(png_ptr, info
23
        png_read_png(png_ptr, inf
24
        png_destroy_read_struct(&
25
        fclose(infile);
26
        return true;
```

- If executing this input results in these source code lines being executed, we consider this input the set of these lines, and ignore it's contents.
- Now, simple set theory allows us to manipulate our corpus in interesting ways.



- Application of our technique requires a very large sample corpus of sample inputs that have been collected autonomously
- We envision small-scale crawling of the public internet to collect the corpus, for example, HTTP responses can be collected by crawling public HTTP servers.
 - I used this technique to discover MS08-045 and MS09-046, both very old bugs that had evaded other fuzzers, as well as numerous other bugs.
- For our initial implementation, we tested some image decoders using a trivial LWP::Simple crawler.
- Using this data, we infer data about the protocol being tested.



- Example: Internet Explorer Retry With Vulnerability
- While crawling the public internet for HTTP response samples, an IIS machines responded with 'HTTP/1.1 449 Retry With', which included the HTTP Response Header 'MS-Echo-Reply'.
- At the time, searching for any related documentation drew a blank –
 this feature was essentially undocumented, but when included in our
 corpus, the coverage score for internet explorer increased several
 points.
- Trivial mutation of the input revealed an easily exploitable condition, when the response was truncated an object was free()d, but a reference remained to it from another object.
 - Causing JavaScript to request a similar sized buffer assigned the freed buffer to me, where I was able to easily redirect execution.



Internet Explorer Retry With Vulnerability

- This bug was very old, and had existed since at least IE4-IE8.
- I believe this bug had managed to evade other fuzzers, simply because they were not seeded with the data required to find it, despite lots of effort to fuzz HTTP.
 - Our technique allowed us to explore this functionality and identify when we were hitting new, potentially broken, code.
 - I've discovered multiple similar vulnerabilties in a number of other products, including other Microsoft products, but these remain unpatched.



Libpng invalid free vulnerability

 Another nice vulnerability that we found using this technique was CVE-2009-0040, libpng prepared an array of pointers to row data like this:

```
for (row = 0; row < info_ptr->height; row++)
info_ptr->row_pointers[row] = png_malloc();
```

- If an allocation error occurred (for example, insane image dimensions), libpng would attempt to clean up and free all of the row pointers, even the uninitialised ones.
- The uninitialised data in row_pointers[] was easily controllable by decoding a "primer" image, essentially resulting in free(arbitrary).
- Rather than exploiting directly and attacking the system allocator (hard), I was able to free another interesting object unexpectedly, and then get it assigned to a javascript allocation I controlled.

Google Confidential 16

- We found that simple set cover minimisation can be used to great effect exploring and testing the software attack surface.
- Rather than treating program inputs as a stream of octets (Miller et alstyle fuzzing), or simple data blocks (block based fuzzing), we treat them as a set of elements from the finite universe of source code lines from the program to be tested.
- We now simply calculate the cardinality of our large corpus, and then attempt to find the smallest sub-collection such that the union of those inputs has the same cardinality.
- Obviously set-cover minimisation is NP-hard, however a simple nonoptimal approximation is trivial.



- Our initial results with Corpus distillation were encouraging, we were able to break some high profile software and find very old bugs that others had missed.
- Just simple mutation of our distilled corpus would break most software (or a corpus distilled using coverage data for program A would break similar program B without modification!)
- Using a combination of corpus distillation and flayer produced yet more breakage, we were able to rapidly cut the time required to use flayer effectively and avoid the overhead of constraint solving.



Corpus Distillation

- Interestingly, this proved to be a good way of validating that all of the edge cases handled in implementation A, were also handled in implementation B.
- In multiple cases we were able to break a new implementation by trying testcases that hit specific checks in one implementation, if the authors of another implementation had not considered this case, it would often crash.
- I have dozens of cases where building a corpus with an open source implementation would crash every proprietary implementation I could find.
- Unfortunately most of these bugs are still unpatched.



Deep Coverage Analysis

- Despite good results from Corpus Distillation, we felt that basic block based coverage was holding us back.
- It's clear that certain constructs, such as CRCs, are unlikely to be maintained without some form of protocol definition the fuzzer can refer to.
- We solved this problem by introducing sub-instruction profiling.
 - Existing coverage-driven fuzzers at best use basic blocks for determining code coverage, but we felt this was still too high level.
 - Of course coverage data derived from basic blocks is equivalent to instruction level coverage, but it's easy to imagine how a large amount of logic can be encoded in a single instruction.



```
#include <string.h>
int main(int argc, char **argv)
     return strcmp(argv[1], foobar );
           F3 A6 REPZ CMPS BYTE PTR DS:[ESI], BYTE PTR ES:[EDI]
```



strcmp, memcmp, and similar

- Any reasonable compiler will inline the string comparison to a single machine instruction, a rep cmps.
- This can be optimised into an entirely branch-less subroutine, and thus coverage information is highly misleading.
 - Basic block based coverage can hide large amounts of program logic
 - Feedback driven fuzzers that uses basic block execution counts are unlikely to ever proceed past checks like this, unless this constant happens to be pre-populated as part of a block definition or protocol grammar.



```
#include <stdlib.h>
int main(int argc, char **argv)
     return atoi(argv[1]) == 0xabcdef;
           3D EF CD AB 00 CMP
                            EAX, 0xABCDEF
           OF 94 CO
                 SETE
                            AL
```



Arithemetic, Immediates and Constants

- Even simple arithmetic operations may be hiding significant program logic.
- Unless a constant like this is pre-seeded, random mutation is unlikely to discover it.



Solution

Sub-Instruction Profiling

- We solve this problem using sub-instruction profiling, essentially using DBI to instrument common code patterns that may shield hidden logic.
 - This is relatively straightforward using PIN, which tells us whenever a new basic block is encountered so that we can examine it and install instrumentation data.
 - We insert calls before and after the interesting code points, and then calculate a new "deep" coverage score.
 - Consider the first example, we can improve feedback by installing instrumentation that complements coverage by examining the value of ecx before and after the rep cmps.
 - The 32bit immediate comparison can be broken into 32 bit-sized chunks, and we can assign a score based on "depth" reached.



Deep Cover

Reconstruction literals, immediates, etc

- This technique has proven to be extremely useful. An interesting case study involved attempts at attacking various PNG decoders.
- In several cases, we've found bugs that required a chunk to have a correct crc32 present just using simple mutation.
 - We break the crc32 comparison into bit-sized manageable chunks.
 - Originally we used the inverted hamming distance between source and destination as the coverage score, but this proved unexpectedly susceptible to local minima.
 - Now we simply count the correct bits starting from the MSB until an incorrect bit is encountered.
 - The feedback received from this instrumented comparison is now enough to allow the fuzzer to reconstruct the correct crc with zero knowledge of the algorithm.

Google Confidential 26

Sub instruction Profiling

Sub instruction Profiling vs. Constraint Solving

- Perhaps the classical solution to similar problems is constraint solving (SAGE, fuzzgrind, others).
- We've found that sub instruction profiling combined with simple stochastic hill-climbing has proven to be a more practical solution that has performed equally well.
- We've experimented with both, our work on Flayer and other tools have allowed us to evaluate different techniques.
- While constraint solving does appear to be a more elegant solution, practical experience suggests that it is performs poorly and produces no-better results.



Deep Cover

Making Programs Dumber

- "Deep Cover" is our implementation of sub instruction profiling.
- Currently we use PIN, but now that DynamoRIO has been made available under a more favourable license, we have begun to port it to this new framework.
- We've been able to use this technique to make large amounts of complex logic essentially simpler, more fuzz-friendly chunks, which we've been able to break using surprisingly simple mutation.
- We've been able to eliminate constraint solving, which we consider a major bottleneck in lots of current research.



Flayer

Taking this idea to the extreme

- Flayer is a fuzz framework based on Valgrind/VEX.
- Flayer takes the idea of program simplification to the extreme, essentially stripping away protocol structure and complexity.
 - By extending the "definedness" check implemented by memcheck, we taint all attacker controlled input and trace its flow throughout the target application.
 - Flayer takes your regular application that parses some complex data, and makes /dev/urandom an effective fuzzer, regardless of what protocol or format your program reads.
- We published this idea at USENIX WOOT, and others have since extended the idea. We really believe in this idea, and think there is some exciting potential here.



Making Programs Dumber

Flayer

- Flayer taints user input and traces it's flow through an application with bit precision, flayer monitors when a tainted condition is tested, and controls whether the path is taken or not.
 - Thus, flayer knows when an application makes a decision based on something an attacker provides.
 - Using some simple heuristics we can decide if an attacker could have taken this codepath, and force it to be explored regardless of whether the input would have caused it.
- We've used this technique to uncover major vulnerabilities in lots of software, such as openssl, openssh, libtiff, libpng, etc. Flayer strip's away the underlying protocol structure, making 'sshd < /dev/urandom' an effective fuzzer.





Tavis Ormandy Google, Inc. taviso@google.com