CS4110 Software Testing and Reverse Engineering

Olaf Maas

February 2016

1 Firmalice [1]

The number of devices running embedded software is growing. As these devices often have a security critical function it is important there are no security flaws in the firmware. Analyzin security flaws is a time consuming job. Wang et al. present Firmalice, this is a system which is able to automatically detect Authentication Bypass Vulnerabilities (Backdoors) in the firmware of embedded devices.

The process of analyzing a system can be divided in 6 steps. The first step is loading the firmware into the system. In the second step the security policy must be set. This must be done by a human as backdoors are logic flaws and they depend on the intended policy a programmer had. It is hard to reason about this so a human should intervene in this case.

After this the software is statically analyzed. During this step Firmalice creates a program dependency graph of the firmware. It then starts at a privileged execution point and generates a backward slice of the firmware to the entry point. This slice is passed to the next step the Symbolic Execution step. In this step firmalice tries to find a valid path from an entry point to a privileged program point. When such path is found this path is passed to the authentication bypass check. This module checks if the path is a backdoor. If the state can be achieved using prior communication to the device firmalice produces a function that produces the valid input based on output.

Firmalice was run on three different firmwares with known firmware vulnerabilities. In two of them it detected the vulnerability, in the third sample the authentication bypass could only be used if a user was already authenticated and so firmalice did not detect this. The run time for these examples lied between 30 minutes and 12 hours.

2 Mayhem [2]

Mayhem is a program used to automatically find exploits in binary programs. It is able to take input from various sources such as the network, environment variables and files. For every bug found Mayhem automatically generate by a working shell spawning exploit.

Mayhem introduces the concept of hybrid execution. Normally a symbolic executor reasons about 1 program path at the time or fork at each branch. The first is fast, but statements are executed multiple times, the latter consumes a lot of memory. Mayhem combines the two concepts. To achieve this Mayhem has basically two processes. The first process is the Concrete Executor Client which executes code natively on the CPU. The other service is the Symbolic Execution Service, this service symbolically executes the blocks the Concrete Executor Client sends and outputs test cases, crashes and exploits.

The second concept introduced by Mayhem is an index-based memory model. A fully symbolic memory model has scalability problems and mayhem solves these problems by making the memory partially symbolic. Writes are always concretized, symbolic reads however are not.

Mayhem found 29 exploits in different binary software. It took mayhem 2 seconds up to an hour of execution time to come up with an exploit.

3 Fuzzing with code fragments [3]

In order to apply fuzz testing on an interpreter, it would be useful if the fuzzed input is a semantically valid program. If the fuzzed input were random, only the lexical and syntactical analysis of the interpreter would get tested. These parts are simple compared to other parts of an interpreter, such as code transformation, just-in-time compilation, or execution of the code. Holler et al. created a framework called LangFuzz, which allows black-box fuzz testing of engines based on a context-free grammar. LangFuzz consists of roughly three parts: code mutation, code generation, and environment adjustments.

Code mutation consists of two phases: a learning phase and a mutation phase. During the learning phase, sample code files is parsed using the language's grammar. During the mutation phase, code fragments from a single file are randomly replaced with other fragments seen during the learning phase. Many mutants are created from the original sample files.

Code mutation can only learn from old code. Code generation focuses on producing new code, possibly generating code typically not seen in the mutation approach. Using the language's grammar, a random walk over the tree of possible expansion series is performed.

When fragments are replaced, the new fragments might not be semantically valid. For example, in some languages variables need to be declared before they're used. In order to adjust for the environment, identifiers are replaced with identifiers occuring somewhere in the rest of the program.

To test the performance of LangFuzz, it was run in parallel with jsfunfuzz, a fuzzer for javascript interpreters. LangFuzz found a large number of bugs that jsfunfuzz could not detect, and vica versa. Therefore, it is recommended that LangFuzz is used in conjunction with jsfunfuzz. In conclusion, LangFuzz is an effective tool in finding defects in interpreters.

4 Taming compiler fuzzers [4]

Fuzzers are very effective at finding compiler bugs. However, because fuzzers use random test cases, sometimes a particular bug is triggered multiple times. Some of the bugs found are also non-critical or known bugs. Chen et al. describe the fuzzer taming problem: given a large collection of random test cases that trigger bugs, rank them such that distinct bugs appear early in the list. For evaluation, the expected bug discovery curve without any fuzzer taming is given as a baseline.

To tackle this problem, they define a distance function between test cases. Many different distance functions were tested, such as the edit distance between compiler output, or the euclidean distance between tokenized test cases.

Using this distance function, they then sort test cases using the first point first (FPF) technique. They also attempted to cluster similar tests and pick a test from a cluster to create a ranking. However, in experiments this approach turned out to be more complex to use, more computationally expensive, and less effective.

Test-case reduction, reducing the size of large tests while still inducing the same failure, was also found to play a big role. Without test-case reduction, it was difficult to improve upon the baseline.

By taming the fuzzer, the same distinct bugs can be found several times faster than when a user inspects test cases at random.

5 How We Get There: A Context-Guided Search Strategy in Concolic Testing [5]

Concolic testing can be exhaustive. Since covering every branch can take too long for even medium sized programs to complete within test budget. search space is one of the biggest challenges in concolic testing. Concolic testing is based on symbolic values instead of concrete values. The symbolic execution maintains a symbolic memory state and symbolic path, which represents states along the execution paths. At the end of an execution path a vector with concrete values can be calculated with a solver and tested against the code.

The idea of concolic testing is testing all the branches and getting coverage for every execution path, the execution tree. There often too many branches also called path explosion too select and this causes problems to solve the all the branches in reasonable time.

Strategies for concolic testing the paper discuss are using random path execution, CarFast greedy algorithm that looks for paths that will get a high code coverage. CFG Directed search, that makes decisions based on the distance of other paths and solving close by branches first. Generational search, this algorithm takes multiple branches at once and create multiple vectors. The path with the highest coverage will be used to create next generation.

CGS - Context guided search: CGS looks at examines a branch in the execution tree and decides to select or skip it. It does this by checking if a context is already examined. For each branch it calculates multiple k-contexts. The k means the branches that are k or more away from the root of an execution path. CFG stores these contexts in a context cache to able skip them later if it finds a matching context.

Context: All nodes picked from a branch that precede the root of the path with k distance form the context. Max k can be a set number or CFG can increasingly choose a higher max k. The optimal k may very depending on the size of the program and the testing budget.

Dominator: If certain nodes need a specific path to reach it, we call the nodes it depends on dominant. The route through these dominant nodes will always give the same context, so CFG tries to remove these contexts that from the general context.

6 Symbolic execution for software testing: three decades later [6]

Symbolic execution uses symbolic values instead of concrete values as input. The values are used to test execution paths in the program. The goal is to create a set of inputs with the symbolic values that every execution path can tested once.

Concolic testing: Performs symbolic execution dynamically while executed. it maintains a concrete state and symbolic state. the concrete state maps all variables to concrete values, while the symbolic state maps variablees that have non-concrete values. The mapping of the concrete state requires concolic testing to have random initial input values. During the execution it collects the symbolic constraints from statements it encounters. At the end of a run it solves these statements and steers the next execution path by using these solved concrete values.

Path explosion: The amount of paths increases exponentially with the amount of brnaches in the code. In order to decrease the amount paths the program has to test it filters path that do not depend on symbolyc input and path that are infeasible with the path constraints. Another option to solve this problem is using search techniques to steer the execution paths.The search heuristics focus for instance on high statement or branch coverage. Random branch picking also has positive results. Using a fitness function, like evolution, to decide which path to solve also works good. Another approach to solve path explosion is to decrease the complexity of path exploration. Techniques that are user are: merging of branches, reusing part of paths or lazy test generation.

Constraint solving: Solving the constraints with the symbolic values found during the execution, has an large impact on the runtime. Solution to this can be to eliminate certain parameters in a constraint or to use caching to solve (parts of) already solved equations quicker.

References

[1] Yan Shoshitaishvili et al. “Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.” In: *NDSS*. 2015.

[2] Sang Kil Cha et al. “Unleashing mayhem on binary code”. In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 380–394.

[3] Christian Holler, Kim Herzig, and Andreas Zeller. “Fuzzing with code fragments”. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 445–458.

[4] Yang Chen et al. “Taming compiler fuzzers”. In: *ACM SIGPLAN Notices*. Vol. 48. 6. ACM. 2013, pp. 197–208.

[5] Hyunmin Seo and Sunghun Kim. “How we get there: A context-guided search strategy in concolic testing”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 413–424.

[6] Cristian Cadar and Koushik Sen. “Symbolic execution for software testing: three decades later”. In: *Communications of the ACM* 56.2 (2013), pp. 82–90.