International Journal of Secure Software Engineering
Volume 7 • Issue 1 • January-March 2016

Steps Towards Fuzz Testing in Agile Test Automation

Pekka Pietikäinen, Oulu University Secure Programming Group, Oulu, Finland
Atte Kettunen, Oulu University Secure Programming Group, Oulu, Finland
Juha Röning, Oulu University Secure Programming Group, Oulu, Finland

ABSTRACT

Including and automating secure software development activities into agile development processes is challenging. Fuzz testing is a practical method for finding vulnerabilities in software, but has some characteristics that do not directly map to existing processes. The main challenge is that fuzzing needs to continue to show value while requiring minimal effort. The authors present experiences and practical ways to utilize fuzzing in software development, and generic ways for developers to keep security in mind.

KEYWORDS

Continuous Integration, Fuzzing, Instrumentation, Secure Development Lifecycle, Security Testing

1. INTRODUCTION

Software security vulnerabilities are not a new phenomenon. With the increasing amount of digitalization in society, the impact of vulnerabilities becomes more threatening. For example, web browser vulnerabilities have resulted in widespread intrusions and the existing exploit kits continue to be a cost-effective means of installing malware. Techniques such as data execution prevention (DEP) and address space layout randomization (ASLR) make exploitation of bugs more difficult than before, and sandboxing techniques are becoming a widely used means of isolating code that processes potentially malicious data. While these advances help in making exploitation significantly more difficult than it was a decade ago, they do not make it impossible. The problem of finding and fixing implementation level security issues has remained largely unchanged from what it was a decade or even two ago, mainly due to the same programming languages being used for writing software.

The process of generating software, however, has changed. Traditionally, software development constituted of the distinct phases of requirements gathering, design, implementation, verification and maintenance, which are executed as a waterfall or, recently, in a more iterative fashion. Adding security activities to these steps is quite straightforward, as done by the Microsoft Secure Development Lifecycle (Microsoft, 2012). Agile and lean software development performs the same phases more or less concurrently. Unfortunately, trying to perform the same security activities in an agile process

DOI: 10.4018/IJSSE.2016010103

reduces agility, and thus the activities need to be relaxed or they will not be done (Beznosov & Kruchten, 2004; Keramati & Mirian-Hosseinabadi, 2008).

Fuzz testing (fuzzing) is a practical way of testing software for security vulnerabilities arising from processing external input and is usually included in secure development lifecycle models. In 2015, nearly all high-impact vulnerabilities with a Common Vulnerabilities and Exposures (CVE) entry were found with fuzzing (Zalewski, 2015b). Fuzzing is clearly something that should be done when developing secure software, but applying it efficiently as a part of an agile process can be challenging.

A major challenge comes from test automation. Continuous Integration and Delivery (CI/CD), Test Driven Development (TDD) and Behavior Driven Development (BDD) all aim to provide mechanisms for delivering working software quickly, in small increments. Fuzz testing campaigns do not directly map to the used workflows, and thus can easily be seen as a complicating add-on by developers.

Another difficulty comes from lack of suitable personnel. Teams need to test, instrument and correctly interpret the results of fuzzing in a multitude of environments, test legacy code, previous features as well as new functionality. Including a security expert in each development team is infeasible. To overcome this, organizations often have a central "Software Security Group", which can be overburdened and difficult to scale to the needs of different project teams. In this paper, we describe experiences in utilizing fuzzing as a part of a software development process, and present practical ways to utilize fuzzing in software development, and present generic ways for developers to keep security in mind.

This paper is structured as follows. Next, we describe the general process of fuzzing and how the parts can be automated and mapped to agile software development practices (Section 2). Then, we describe existing secure development lifecycles and how they consider fuzzing (Section 3). In Section 4, we describe experiences on the difficulties involved in utilizing fuzzing and how we have attempted to overcome the difficulties using two examples. We discuss the findings in Section 5. Finally, conclusions are drawn.

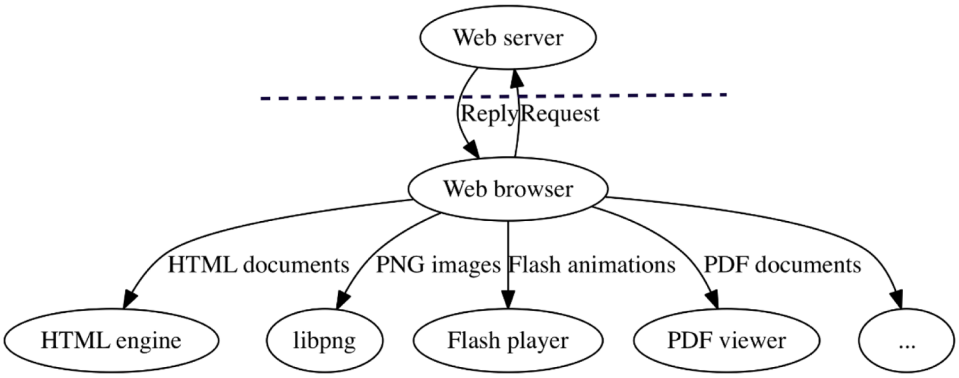
2. FUZZING

Fuzz testing originates from the late 1980's, when Professor Barton Miller discovered that modem line noise (i.e., a stream of random bits) could cause faults in commonly used UNIX tools, and began investigating the phenomenon (Miller, Fredriksen, & So, 1990). The work was continued at the Oulu University Secure Programming Group (OUSPG), where the PROTOS project developed methods for model-based robustness testing and enabling the industry themselves to find vulnerabilities (Kaksonen, 2001).

Fuzzing generates invalid or random inputs and injects them into a program. This technique can be used as such with very simple instrumentation to perform black box testing, usually combined with relatively simple heuristics for detecting obviously erroneous program states, such as fatal signals, or in a more white-box fashion, such as runtime program tracing to gain insight, which can be used to control the data generation.

One of the important properties of fuzzing is that one can start testing a program with very little knowledge about the target, and gradually refine the testing by improving the test cases and instrumentation. As an additional benefit, each found issue comes paired with a proof of concept input that can trigger the error, which proves that the bug can be triggered externally. This is unlike static analysis tools, which locate bugs in software, but do not easily demonstrate how the affected code can be reached in practice. This is especially important for programs like web browsers, where nearly all input is from an untrusted, potentially malicious, source.

Figure 1. Simplified threat model of web browser



Fuzzing consists of the following phases (Sutton, Greene, & Amini, 2007):

1. Identify target
2. Identify inputs
3. Generate fuzzed data
4. Execute fuzzed data
5. Monitor for exceptions
6. Determine exploitability

These phases can roughly be mapped to agile software development practices, which we will now describe.

2.1. Identifying Attack Surface through Threat Modelling

The target, obviously, is the software under development or an external dependency of it. The targeted component can typically be reached through a number of interfaces, such as "HTTP request over network" or "Read PNG file from local file system", each using some more or less defined protocol. These are identified during threat modelling. Threat modelling can be done using a variety of methods, e.g., the Microsoft STRIDE model. (Hernan, Lambert, Ostwald, & Shostac, 2006).

In STRIDE, a data flow diagram (DFD) of the system is built. The DFD includes the program components, data flows between them and trust boundaries. For each component, applicable threats are characterized into six categories: Spoofing of user identity, Tampering, Repudiation, Information disclosure, Denial of Service, and Elevation of privilege. Fuzzing is applicable to data flows, where the data could be tampered with. Usually the effect of test cases is seen as a Denial of Service (crash), but can be any of the categories.

The aspects of the threat model that are interesting from the viewpoint of fuzzing are inputs crossing trust boundaries and the components processing them. If the input comes from a completely trusted source, i.e., all protocol layers are fully trusted, fuzzing the input is not necessary. If the component is run with high privileges or is implemented in a programming language known to be susceptible to memory safety issues, namely C and C++, the risk for that input is high.

An example of a DFD that could be used as a basis for fuzzing is shown in Figure 1, which shows how a web browser interacts with other components. It sends requests to web servers. The server responds with content the browser processes internally or using external libraries or plugins. The content crosses a trust boundary, from the untrusted Internet to the browser process, which has access to, e.g., operating system facilities and data owned by the user who runs the browser. A more

complete threat model would show these, as well as sandboxes used to isolate components that are known to have risks, such as external plugins.

Protocols that have a context-sensitive grammar, e.g., include length fields, are more susceptible to the types of defects that can be found with fuzzing, whereas regular languages are easier to parse correctly (Bratus et al., 2014). Unfortunately, formal language theory is not likely to be within the skillset of developers performing threat modelling.

Threat modelling produces a list of interfaces to fuzz, which can be prioritized and placed on the product backlog.

2.2. Test Case Generation and Execution

There are several approaches to test case generation in fuzzing. It is usually divided into two categories: generation-based fuzzing and mutation-based fuzzing, both of which have benefits and drawbacks.

Generation-based fuzzing (Kaksonen, 2001) is based on a fixed model of the input space, which is then used as a basis for building test cases. In some cases, this can be done directly from source code of the software to be tested, but often must be done separately. The method works very well with protocols and file formats that have a formal specification, from which the model can be inferred. While the model is often tree-structured, as are the protocol definitions, the model can also include fields, such as checksums, which are used by implementations to determine whether the input is valid. One of the important factors of exact models is that essential fields such as these can be automatically filled correctly making it much more likely that the target program does indeed process the contents of the test case instead of just dropping it as invalid. The main downside of generation-based fuzzing is that developing a comprehensive test suite including all the models is a major manual effort, as the test suite is essentially a minimal implementation of the software to be tested.

If the specification changes during the project, as it is often the case in agile projects, so must the model. Therefore, they are mainly of use for parts of the project where the model remains constant, or which are considered critical enough to justify building or purchasing a test suite.

Mutation-based (or template-based) fuzzing operates based on samples of data produced by valid implementations. Most mutation-based fuzzers operate by making simple changes to the data without any knowledge about the semantics. Typical mutations include flipping bits at random, omitting data, repeating data and writing random data somewhere. Even though they lack the finesse of generation-based fuzzers, these black-box tools are important due to their ease of use and the fact that they can be made general purpose. Even the drawback of not understanding the exact protocol semantics can be worked around, e.g., testing against a system under test (SUT) that has checksum verification disabled makes it possible to improve test coverage without adding support for calculating the checksum into the fuzzer.

Radamsa (Helin, 2015) is a collection of mutation-based fuzzers, which has the primary objective of being as simple to use as possible. Unlike most similar tools, it is intended to work for just about any kind of data without any extra configuration. It does some simple heuristics on the input data (binary or ASCII), and the operation of which ranges from trivial common mutations to more novel ones, which often approximate the operation of generation-based ones.

Hybrid approaches also exist. Bekrar, Groz, and Mounier (2012) propose a grey-box approach to fuzz testing which is based on defining vulnerability patterns at assembly level before the actual fuzzing process. The idea of these patterns is to identify potentially vulnerable code in the binary, for example functions that could lead to memory corruption, and represent them as models. Next, taint analysis is applied by marking all potentially dangerous data as tainted and tracking their propagation during execution. Thus fuzzing can be improved by selecting the most promising test sequences that are likely to trigger potential vulnerabilities and restrict the test space. The actual fuzzed data is then generated by applying mutations on sample inputs or by modelling inputs using some learning methods or the target specification. The execution of the program is observed using systematic path exploration technique, called concolic or symbolic execution, where the constraints tied to branch instructions are

Table 1. Comparison of fuzzer types

	Template-based	Mutation-based	Hybrid	Coverage-based
Requirements	Specification	Samples of data	Few samples	Samples
Source code required	No (can be used as specification)	No	No	Yes
Effort required	High	Very low	Low	Low
Examples	PEACH, Codenomicon	Radamsa, zzuf	fuzzgrind	afl, covFuzz
Popularity	Medium	High	Low	Emerging

extracted and solved in order to explore new paths and discover new potential vulnerabilities. These techniques have not gained widespread use in the industry (Zalewski, 2015b).

A new, practical, approach to fuzzing is done by American fuzzy lop (afl) (Zalewski, 2015a), which employs compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. The fuzzing process is iteratively evaluated using code coverage techniques and search algorithms to improve the test case population and expand the coverage. The mutation algorithms used are purposefully simple to maximize performance, which can be thousands of cases per second. In contrast, individual test cases in model-based fuzz testing of networked protocols may take several seconds. If source code is not available, a black-box "QEMU user emulation" mode can be used, but there is a significant 2-5x impact on performance.

The different approaches are summarized in Table 1. Comparing the effectiveness of the different approaches is difficult. In addition, benchmarking based on known issues in a given code base easily results in a pesticide effect: performing a certain type of testing makes the test targets more immune against that particular type of testing in the long run (Beizer, 1995). In the end, finding the vulnerabilities before they are exploited is what matters. Any new technique that minimizes spent resources while also making useful results has thus by definition been useful. Approaches that are technically simple to implement and able to find relevant issues may only be able to report their findings in a way that requires significant human sophistication to understand and classify, and thus are not, in the end, useful (Bessey et al., 2010).

When the best fuzzing strategy for a given interface is identified, a task to automate it can be placed in the backlog. Depending on the interface and chosen tools, this task may take anything from minutes to months. The simplest programs to fuzz are command-line based tools taking an input filename as a command line argument. These only require a fuzzer and some sample input files to get started. On the other end of the spectrum, the tested interface may be an encrypted network connection where a stateful cryptographic handshake is needed to reach a state where the test case can be injected. In this kind of scenario, sample collection, injection and test automation (or building a model) become much more involved.

2.3. Instrumentation and Analysis

The aim of fuzzing is to find undesired behaviour in software. Crashing is obviously such, but the effects of fuzzing can also be very subtle (Takanen, Demott, & Miller, 2008), which is commonly known as the test oracle problem (Barr, Harman, McMinn, Shahbaz, & Shin Yoo, 2015). An example of this is the Heartbleed (*Heartbleed vulnerability,* 2014), where the response to a fuzzed test case was an abnormally large reply, which contained sensitive information, such as private keys.

For the common case of detecting crashes, building a test oracle is relatively simple, as current operating systems and compilers provide good facilities for it. The ideal output is an uncorrupted

crash dump, which pinpoints the exact location in the code where the problem occurred. The fuzzed test case could cause severe memory corruption, so the program should notify the user as soon as anything abnormal occurs. Memory debugging tools, such as AddressSanitizer (Serebryany, Bruening, Potapenko, & Vyukov, 2012), MemorySanitizer (Stepanov & Serebryany, 2015) and Valgrind (Nethercote & Seward, 2003) help to ensure that the error is detected once it occurs, before corruption occurs. These tools are designed to reveal different types of errors, and have different impact for the performance of target program, so the possibility of using different instrumentation tools during testing should be considered.

For detecting other types of failures, a model of SUT behaviour is needed. Barr et al. (2015) provide a comprehensive survey on the currently used approaches. For fuzzing purposes, the most trivial approach is valid case instrumentation, where a valid operation is performed on the interface. If the interface does not reply with a known valid reply, the test case is assumed to have caused a failure. Unfortunately, this approach only finds the most catastrophic failures (Takanen et al., 2008).

When a suitable test oracle is developed, the question of integrating it into the software development process remains. Undesired behaviour needs to be transformed into an issue, which is fixed by a code commit. Furthermore, a regression test is added as a unit test, and the fix has to be released and communicated to customers.

For a static model-based test suite this is not an issue, i.e., "Test #6423 fails" maps to commonly used workflows. If the input has a random element, as in mutation-based fuzzing, the same issue can most likely be found with several similar inputs. The root cause is the same, so only one issue should be filed. The issue should contain a fully reproducible, minimized, test case. The impact of the defect can be automatically analysed (NULL pointer dereference, stack overflow), but the results are not completely reliable (Microsoft, 2013).

3. FUZZING AS A PART OF AGILE SECURE DEVELOPMENT LIFECYCLES

In response to the growing impact of software vulnerabilities, a number of efforts to improve software development processes to mitigate the effect of vulnerabilities have emerged. Each company has its unique set of business, technology and cultural constraints, which affect the set of activities that are the most beneficial.

Building Security In Maturity Model (BSIMM) is a framework for measuring software security initiatives. Instead of being a guide for secure development, it is based on real-world data from sixty-seven real software security initiatives from a variety of companies. BSIMM can be used to determine the maturity of a given initiative. In BSIMM, fuzz testing is considered as an intermediate (Level 2) or advanced (Level 3) topic in security testing. (McGraw, Migues, & West, 2013)

OpenSAMM (Chandra, 2009) and SAFECode Fundamental Practices (SAFECode, 2011) are prescriptive frameworks for measuring the maturity of security initiatives. OpenSAMM does not explicitly cover fuzzing, but includes automated testing tools as a Security Testing activity. SAFECode provides specific guidance on fuzzing, including suggestions for tools.

Maturity models for fuzzing have also been developed. Codenomicon proposes a maturity model where fuzzing activities of an organization are mapped from a scale of Level 0 (no fuzzing done) to Level 5 (Optimized). Higher maturity levels require, e.g., running more test cases, longer test periods, improving, and optimizing test instrumentation, using several fuzz testing techniques and tools per technique and documentation. (Knudsen & Varpiola, 2013)

In addition to frameworks for measuring the maturity of security initiatives, there are the initiatives themselves. Microsoft is one of the front-runners in this area, and has a well-known secure development lifecycle that has been in development since 2002. During this time, it has evolved address new threats and software development practices, including agile. Fuzzing is a mandatory activity, and is based on threat modelling, and the relative exposure of different entry points. Fuzzing is based on an optimized (based on code coverage) set of templates, and tests are run for a minimum

number of iterations (500,000) plus at least 250,000 iterations since the last security bug uncovered through fuzzing. If fuzz testing does not reveal any crashes, the test campaign is considered as probably inadequate, and a task to figure out possible reasons is started. The reason can also be that the software actually is robust. (Microsoft, 2012)

Generally, the Microsoft model is considered as comprehensive (scoring very high in the maturity models), but too heavy for most organizations. It uses bucket security-related activities, which are performed as a sequence across development increments. This delays the release of software increments, where all activities have to be done for all of the changes that were made.

A common way of making security activities visible in Agile is through security user stories (SAFECode, 2012; Siiskonen, Särs, Vähä-Sipilä, & Pietikäinen, 2014). The security of software increments can be ensured using security assurance cases (Othmane, Angin, Weffers, & Bhargava, 2014).A minimal set of software security activities proposed by Vähä-Sipilä (2014) is:

1. Security risk analysis: an activity that attempts to determine who might want to attack your software, why, how, what would be the impact, and how this risk could be mitigated. This is often referred to as threat modelling or architectural risk analysis. As a result, you should expect to see new and changed use cases, requirements, constraints, and test cases.
2. Setting and enforcing a set of development time constraints and guidelines: These activities can vary from mandating the use of a specific framework to using static analysis to identify bugs in code. These require understanding how your technology choices (e.g., choice of language or framework) contribute to potential vulnerabilities.
3. Security testing: Positive testing for security controls and negative testing that probes the misuse cases of software. This can be exploratory or automated, examples being freeform manual penetration testing and fuzz testing. This should get its input from the risk analysis.
4. Vulnerability management: a set of activities that ensure that your company hears about vulnerabilities in the code you deliver or use. This addresses things like upgrading and patching vulnerable components.

All of these activities are also relevant to fuzzing. In addition to testing itself, risk analysis forms a basis for designing fuzz test campaigns. Certain development time constraints make software easier to test, and fuzz test. Finally, fuzzing results in the discovery of new vulnerabilities, which have to be managed.

The drawback of the previously presented approaches is that they give very little detail on how to introduce fuzzing, and how the introduction of fuzzing is seen inside the software development organization. A common minimum requirement is a data flow based threat model, which serves as a risk-based list of interfaces for fuzzing. Several hundred thousand cases are needed to have basic assurance on the robustness of each interface. If test case injection is slow, even this may be difficult to justify. Finally, evidence that testing has been done for each software increment is required.

In the next section, we describe practical experiences related to the introduction of fuzzing, which answer some of these drawbacks.

4. PRACTICAL EXPERIENCES

In this section, we describe practical experiences associated with fuzzing. We do this from the viewpoint of a research group, whose purpose is to study, evaluate and develop methods of implementing and testing application and system software in order to prevent, discover and eliminate implementation level security vulnerabilities in a proactive fashion. To do this, we have collaborated with industry by providing them with information on vulnerabilities their products have, as well as tools, whereby they could find the vulnerabilities themselves. Some collaboration efforts have

lasted for several years, and the companies have shared their internal experiences with us, which we summarize (anonymously) here.

4.1. Initial Response to Fuzzing

By far, the biggest hurdle has been getting started. The first experience of fuzzing for many companies is receiving an external bug report that includes a test case (possibly a stack trace) that claims to be a security bug. There is no mechanism for easily verifying the bug or finding the relevant developer based on the test case alone. Often, the severity of the bug is downplayed – a working exploit would be required to treat the bug as critical. Eventually, the bug is fixed and internal interest for fuzzing arises. We share experiences in using our tools and demonstrate how we found the bug using them.

Tools, such as Radamsa, have been built to be as easy to use as possible. If the target does not utilize files as input, a test harness and sample collection need to be implemented, which can be a non-trivial task. This may sometimes prevent fuzzing from being used.

Running a fuzzer against code that has not been fuzzed before typically finds a large number of bugs. With legacy code bases even a quick run may find too many issues to realistically fix in a reasonable amount of time. Going through the large number of bugs requires significant effort from skilled personnel. Eventually, a business decision of fixing the most obvious ones is made, and less critical ones are simply accepted.

Initial fuzzing runs may also uncover only a few bugs. This could be due to high code quality, but also bad test coverage (e.g., due to test cases being dropped because of an incorrect checksum, poor quality samples being used with a mutational fuzzer, the SUT masking all exceptions or lack of instrumentation). Again, skilled personnel are needed to ensure that testing was useful.

4.2. Automating Fuzzing

After the initial results of fuzzing have been incorporated, the next hurdle is automating it and making it a regular part of the software development process. An individual "fuzzing expert" from the team or the SSG often builds the infrastructure. The automation is used for a while, and eventually stops finding new bugs. The expert already has new tasks and the infrastructure is eventually abandoned. In our experience, improving the fuzzer or sample set, or simply recompiling with a memory-debugging tool, would continue to find bugs.

For companies that have mature software development processes, integrating fuzz testing into their test automation can be challenging. Continuous Integration considers each new commit to potentially cause new failures. If a failure is seen, the new commit is blamed. If the test cases are random, as they are in fuzzing, the faulty code could have been introduced at any point. Adding a separate task that performs fuzzing is possible, but the workflow is different. For example, builds used for fuzzing may use different compile-time instrumentation than the normal builds, duplicates need to be handled, and the commit that introduced the bug needs to be found.

4.3. Best Practices

What has worked is placing fuzzing activities (automating the testing of some interface) on the backlog. Fuzzing also can leverage infrastructure from other activities, e.g., UI tests done with Selenium webdriver are used with a fuzzing proxy. The team documents all new JSON API's in a standardized way. API documentation is automatically generated and fuzzed test cases are automatically generated from the model

Agile methods require a "Definition of done", a list of criteria that must be met before a task is considered complete. Defining a number of test cases or the amount of time used for fuzzing is a mandatory minimum, but does not ensure that testing is effective. Coverage-based techniques can be used to find areas that are not tested, but do not ensure effective testing (Inozemtseva & Holmes, 2014). However, in our experience, coverage-based sample set optimization greatly increased the efficacy of mutation-based fuzzing.

Finally, the question of the economics of fuzzing and test automation is always relevant (Ramler & Wolfmaier, 2006). How much resources should be spent on fuzzing and when are we doing enough? What is the opportunity cost of fuzzing, i.e., are would the effort spent on fuzzing be more productive elsewhere? Is it worthwhile to do it in-house, or can it be externalized to a bug bounty program or an external consultant (Finifter, Akhawe, & Wagner, 2013)?

We now present two examples on how fuzzing can be used to find defects. The first, web browser testing, shows how a very advanced and automated process of fuzzing can work. The second, LTE signalling protocols, shows how practical challenges can make even small-scale fuzzing campaigns difficult.

4.4. Example: Web Browsers

Web browsers are the primary means of accessing Internet-based services. Their install base is in the range of hundreds of millions and the market is shared by only a handful of vendors. In the beginning browsers were primarily used for displaying static hypertext and related images. Instead of being viewers of static data, current browsers are essentially JavaScript-based programming environments, with support for processing many kinds of data. To provide a seamless user experience, current browsers automatically download and process nearly any data they are requested to fetch, as visualized previously in the threat model in Figure 1. From a security perspective, this is a nightmare, since this combined with the multitude of data formats supported in modern browsers makes the attack surface enormous. The ease of injection of malicious data and homogenous environment makes them a good target for malware, which combined with the evolving standards and growing consumer expectations make securing browsers an unprecedented engineering challenge.

Sandboxing techniques are used to mitigate the effect of vulnerabilities, but this has merely made exploitation more laborious - bypassing the sandbox, address space randomization etc. is still possible, but may require chaining exploits for several bugs together.

Two major open-source browsers, Chromium (used as the basis for Google Chrome) and Firefox are developed in an agile fashion, with major stable releases occurring every six weeks. Web browsers are also one of the most fuzzed pieces of software, as browser vendors have invested heavily in test automation (Arya, 2015) and developed novel instrumentation methods, such as AddressSanitizer (Serebryany et al., 2012). Also, vendor bug bounty programs and the wide install base have attracted third party security researchers. These factors, and the public nature of development make them an ideal case study for the use of fuzzing in test automation.

Browsers quickly gain new features as web standards evolve. The features first appear in nightly builds. The most invasive and risky features are only enabled, for testing purposes, when a special configuration flag is set. As trust increases in the feature (both functionality and robustness), the feature is enabled by default and included in beta and release versions.

With quickly evolving programs, like web browsers, problems occur with the development of fuzzing test case generators. Generation based fuzz testing requires work and time, when new features have to be researched and new generator grammars have to be developed. With mutation based fuzzing the problem is finding sample files, at least when there is a completely new feature and old samples cannot be recycled.

Because Chromium and Firefox are both open-source projects their test suites are publicly available. As new standards are designed, both vendors start to develop new features to conform to the standard. At the same time, they start to add new tests for that feature. Individual tests in these test suites are commonly designed using a vendor specific design model. With a simple conversion script, the individual tests can be converted to standalone files that can be used with a mutation-based fuzzer when testing the new feature. New vulnerabilities and bugs can be revealed from other browsers by samples derived from another vendor's test suite.

Our group participates in these efforts (Pietikäinen et al., 2011). Since our initial work in this area, finding new vulnerabilities with the same methods has become increasingly more difficult. Increasing

the amount of used hardware has not significantly helped. Due to the efforts of both browser vendors and the bug bounty hunter community the "low hanging fruits" have been picked and we have to be smarter with our fuzzing efforts to reach the same results.

In 2010, we used blind mutation based fuzzing, with random samples downloaded from the Internet, and easily found new bugs. In terms of browser features, most of these random files are outdated. In practice this meant that our fuzzing efforts were directed towards the testing of older features in browsers. These features were more likely to be already fuzzed by browser vendor, or the bug-bounty hunter community. To be more effective in our fuzzing, we needed to balance our fuzzing capabilities to cover the whole threat model of the browser more evenly.

As a solution, we decided to use code coverage tools, like SanitizerCoverage, to preprocess our sample corpus. Preprocessing minimizes redundancy from the sample corpus and thus emphasizes samples that stress rarely seen features in the target. Preprocessing also removes samples that use features that are either deprecated, or not yet implemented, in the target. In practice this method has lead to the discovery of bugs in features like Google Chrome's Content-Security-Policy(CSP), resulting in Chrome bugs 427397 and 430351 (Google, 2015). At that time only 0.002% of randomly collected samples included CSP. Also, we found a vulnerability in Mozilla Firefox MP3 support (CVE-2015-0825). At that time, MP3 was marked as an unsupported format in Firefox documentation.

We also use runtime coverage analysis while fuzzing. Runtime code coverage analysis allows our system to automatically collect interesting mutated cases and feed those back to our fuzzing system for further mutating. This process allows our testing system to reach code paths that would have been unreachable with our previous approaches.

We have now found over 110 security vulnerabilities (CVE number) in the stable versions of the major web browsers. In addition, a similar number of vulnerabilities were found in pre-release (unstable, beta) versions, but the vendor does not analyze the security impact as heavily in these cases.

As another approach to add effectiveness to our fuzzing we targeted the fuzzing into individual third-party libraries used in the browser. For example, these libraries include parsers for different media formats. Direct fuzz testing of these libraries allows us to run more test cases, in a given time, than we could run when testing the whole browser. Of course, as a side effect we can miss some vulnerabilities that are not due to an error in the library itself but in the way the browser handles the library. Hence, testing inside the browser is also required. Using runtime code coverage analysis, when fuzzing third-party libraries, allows us to collect a sample corpus that can be later used as a base for browser fuzzing.

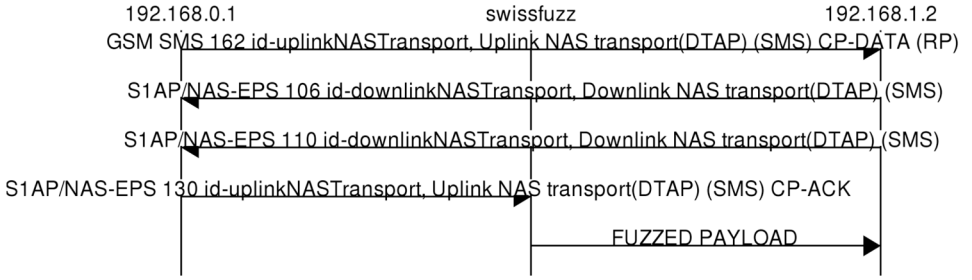
4.5. Example: Network Protocol Fuzzing

When the tested interface is a network protocol, some additional challenges arise. For example, the 3GPP Long-Term Evolution (LTE) standard defines an all-IP architecture for mobile communication. The number of protocols involved is quite high, and model-based fuzzing test suites for most of them do not exist, mostly due to the complexity of the protocols and limited number of potential customers for them.

We were tasked with developing a method for enabling testers with limited security experience to be able to perform some fuzzing on parts of the LTE protocol stack, e.g., H.248/GCP, S1AP, X2AP, RANAP, BSSAP, ISUP, BICC and SCCP.

To accomplish this, we developed swissfuzz, an extension of Radamsa (Helin, 2015) for handling stateful network protocols. It is a Python-based toolkit for collecting samples from network traffic, isolating the application payload from TCP, UDP and SCTP flows, feeding the samples to Radamsa, executing tests and doing valid-case instrumentation. It can simulate being a client, server and a pass-through proxy, which fuzzes traffic when a given state is reached, as illustrated in Figure 2. Traffic between two, unmodified, devices is passed through the fuzzer. When the conversation has reached the desired state, instead of passing the traffic through, a fuzzed packet (based on multiple samples of that part of the conversation) is sent.

Figure 2. Man-in-the-middle fuzzing using swissfuzz



With swissfuzz, stateless fuzzing can be done using only command line tools and packet captures as the samples. For stateful protocols, some Python programming is required.

Technically, the solution met the requirements of the company. Unfortunately, it has not seen widespread use inside the company, as Python is not a common skill for the personnel doing testing, so the usage is limited to one fuzzing enthusiast. A graphical user interface for defining tests for stateful protocols would be required to have it used more widely.

In our own internal use, we use swissfuzz as the first step for testing networked applications. Even a brief test (under one hour) is enough to find suspicious behaviour, and thus a basis for more detailed examination. Examples of bugs found include:

- Update agent for antivirus software consumes large amounts of memory and crashes after fuzzed response from update server;
- Industrial Control System process crashes after sending fuzzed version of traffic normally seen on network;
- Smartphone map application crashes when map graphics from server is fuzzed.

These types of findings clearly demonstrate that testing for low-hanging fruit should always be done.

5. DISCUSSION

This section discusses the challenges in adopting and automating fuzzing and suggests possible solutions.

The process used for testing Chromium (Arya, 2015) demonstrates a very advanced way of utilizing fuzzing, which is supplemented by a bug bounty program. Implementing similar infrastructure is beyond most projects, but many activities can be done with limited resources.

New features are added to the threat model, which in the case of web browsers is essentially a list of new supported formats. The features are phased in gradually, and are not enabled until they have been tested for both functionality and security. Finally, comprehensive feature test suites are reused as models for model-based fuzzers or samples for mutation-based ones. The quality is high enough that reasonably high bug bounties can be paid to the vibrant community of external bounty hunters, while being cheaper than in-house testing.

Fuzzing campaigns are not always successful. The greatest challenge is to firmly establish fuzzing in the organization. The best way of doing this is for developers to constantly see new, useful, results. This encourages further investment in testing, and reduces the risk of the fuzzing campaign being a one-man effort that will eventually be abandoned. This is exactly what occurred in the LTE signalling protocol example.

The effectiveness of fuzzing depends on how well it is executed. This begins from data flow based threat modelling, which should be continuously updated. Threat modelling should also take results from previous fuzzing into account. If new issues are found with fuzzing, the interface should

be a candidate for increased fuzzing efforts. Even limited dumb fuzzing with manual instrumentation can find some low-hanging fruit, and thus should be done for all inputs of the system. The quality of third party components should also be considered.

To stay in use, fuzzing needs to be automated. Even then, the infrastructure will eventually stop finding new bugs. Up to a point, improving test case generation (better models or samples, adding new fuzzers) and instrumentation (use of new type of memory debugger, custom test harness) will continue to find new bugs. Many of these activities only require a small one-time cost, and are easily justified. Others require more significant efforts, including the use of experts.

Coverage guided fuzzing is a promising field, and can improve the quality of fuzz test campaigns. In addition, coverage provides insight into whether placing further effort into improving the test campaign would be useful. There are several ways of measuring coverage, and finding a practical combination is necessary. (Tsankov, Dashti, & Basin, 2013)

A key question is the availability of the skills required to setup an effective fuzz test campaign. The Software Security Group is pivotal, but eventually testing will have to be done by developers with the proper tools and skills. Setting up ways of working to make fuzzing more effective is one way of doing this. This could be, e.g., standardized ways of building instrumented builds and ways of leveraging results from other activities, like automatically reusing API documentation as a model for fuzz tests.

Some basic skills are required from all development personnel. The fuzzing tools should only require skills that the personnel have. Also, personnel need the skill of understanding crash reports well enough to find the root cause of the bug.

6. CONCLUSION AND FUTURE WORK

This paper concludes that fuzzing can be integrated into agile software development and test automation, but there are difficulties. The main challenge is that fuzzing needs to continue to show value while requiring minimal effort. The web browser example shows a way of effectively using fuzzing in agile development.

The workflow used by fuzzing is somewhat different from traditional testing and this needs to be accounted for. Instead of being an activity that responds to the previous development increment, automated fuzzing is a background process that needs maintenance. Currently, this requires some expert knowledge, but better tools could also fill this gap.

Coverage-based methods are an efficient method of improving the test case corpus, thus improving test quality. Ideally, they would also make it possible to automatically direct fuzzing efforts into areas, which have recently changed in the code. This would also serve as evidence that the new functionality has been tested.

Our future work includes studying how to better integrate fuzzing into Continuous Integration workflows; the use of several types of coverage metrics to both improve fuzzed test cases and understand whether testing is effective, and how to target recent changes in the code. In addition, we will try to find generic ways of working that could be easily adopted by the industry.

ACKNOWLEDGMENT

The authors would like to thank Antti Vähä-Sipilä and the companies we collaborate with for valuable comments for this work. This work was supported by TEKES as part of the Cyber Trust Program of DIGILE (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT and digital business).

REFERENCES

Bessey, A., Engler, D., Block, K., Chelf, B., Chou, A., Fulton, B., . . . McPeak, S. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, *53*(2), 66–75. doi:10.1145/1646353.1646374

Arya, A. (2015). *Analyzing chrome crash reports at scale. Nullcon 2015*, Goa.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Shin Yoo. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering, 41*(5), 507-525.

Beizer, B. (1995). *Black-box testing: Techniques for functional testing of software and systems*. John Wiley & Sons, Inc.

Bekrar, S., Bekrar, C., Groz, R., & Mounier, L. (2012). A taint based approach for smart fuzzing. *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)* (pp. 818-825).

Beznosov, K., & Kruchten, P. (2004). Towards agile security assurance. *Proceedings of the 2004 Workshop on New Security Paradigms*, Nova Scotia, Canada (pp. 47-54).

Bratus, S., Darley, T., Locasto, M., Patterson, M. L., Shapiro, R. B., & Shubina, A. (2014). Beyond planted bugs in "trusting trust": The input-processing frontier. *Security & Privacy, IEEE*, *12*(1), 83–87. doi:10.1109/MSP.2014.1

Chandra, P. (2009). Software assurance maturity model. http://www.opensamm.org/download/

CVE-2015-0825. (2015). *Stack-based buffer underflow in the mozilla:MP3FrameParser:ParseBuffer function in Mozilla Firefox.* Retrieved from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0825

Finifter, M., Akhawe, D., & Wagner, D. (2013). An empirical study of vulnerability rewards programs. *Proceedings of the 22Nd USENIX Conference on Security,* Washington, D.C. (pp. 273-288).

Google. (2015). *Chromium bug tracker.* Retrieved from https://code.google.com/p/chromium/issues/list

Heartbleed vulnerability. (2014). Retrieved from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

Helin, A. (2015). *Radamsa fuzzer.* Retrieved from https://github.com/aoh/radamsa

Hernan, S., Lambert, S., Ostwald, T., & Shostac, A. (2006). Uncover security design flaws using the STRIDE approach. *MSDN Magazine,* November.

Inozemtseva, L., & Holmes, R. (2014). *Coverage is not strongly correlated with test suite effectiveness*. *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India (pp. 435-445). doi:10.1145/2568225.2568271

Kaksonen, R. (2001). *A functional method for assessing protocol implementation security*. Technical Research Centre of Finland.

Keramati, H., & Mirian-Hosseinabadi, S. (2008). Integrating software development security activities with agile methodologies. *Proceedings of the IEEE/ACS International Conference on* Computer Systems and Applications AICCSA '08 (pp. 749-754).

Knudsen, J., & Varpiola, M. (2013). *Fuzz testing maturity model (Whitepaper)*. Codenomicon.

McGraw, G., Migues, S., & West, J. (2013). Building security in maturity model (BSIMM-V) (Revision 5.1.2 ed.)

Microsoft. (2012). *Microsoft security development lifecycle (SDL) process guidance - version 5.2*. Microsoft Corp.

Microsoft. (2013). *Exploitable crash analyzer - MSEC debugger extensions.*

Miller, B. P., Fredriksen, L., & So, B. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, *33*(12), 32–44. doi:10.1145/96267.96279

Nethercote, N., & Seward, J. (2003, July). Valgrind: A program supervision framework. *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA.

Othmane, L. b., Angin, P., Weffers, H., & Bhargava, B. (2014). Extending the agile development process to develop acceptably secure software. *IEEE Transactions on* Dependable and Secure Computing, *11*(6), 497–509.

Ramler, R., & Wolfmaier, K. (2006). Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost. *Proceedings of the 2006 International Workshop on Automation of Software Test*, Shanghai, China (pp. 85-91). doi:10.1145/1138929.1138946

SAFECode. (2011). S. Simpson, (Ed.). *Fundamental practices for secure software development* 2nd ed.).

SAFECode. (2012). *Practical security stories and security tasks for agile development environments.* Retrieved from http://www.safecode.org/publication/SAFECode_Agile_Dev_Security0712.pdf

Serebryany, K., Bruening, D., Potapenko, A., & Vyukov, D. (2012). AddressSanitizer: A fast address sanity checker. *Proceedings of the USENIX Annual Technical Conference* (pp. 309-318).

Siiskonen, T., Särs, C., Vähä-Sipilä, A., & Pietikäinen, A. (2014). Generic security user stories. In P. Pietikäinen & J. Röning (Eds.), *Handbook of the secure agile software development life cycle*. Oulu: University of Oulu.

Stepanov, E., & Serebryany, K. (2015). MemorySanitizer: Fast detector of uninitialized memory use in C++. *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (pp. 46-55). doi:10.1109/CGO.2015.7054186

Sutton, M., Greene, A., & Amini, P. (2007). *Fuzzing: Brute force vulnerability discovery*. Addison-Wesley Professional.

Takanen, A., Demott, J. D., & Miller, C. (2008). *Fuzzing for software security testing and quality assurance*. Artech House.

Tsankov, P., Dashti, M. T., & Basin, D. (2013). Semi-valid input coverage for fuzz testing. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, Lugano, Switzerland (pp. 56-66). doi:10.1145/2483760.2483787

Vähä-Sipilä, A. (2014). Security in agile product management. In P. Pietikäinen & J. Röning (Eds.), *Handbook of the secure agile software development life cycle*. Oulu: University of Oulu.

Zalewski, M. (2015a). *American fuzzy lop fuzzer.* Retrieved from http://lcamtuf.coredump.cx/afl/

Zalewski, M. (2015b). *Understanding the process of finding serious vulns.* Retrieved from http://lcamtuf.blogspot.ro/2015/08/understanding-process-of-finding.html

Pekka Pietikäinen received his MSc degree from the University of Oulu in 2001. He did his MSc thesis at CERN with the title "Hardware-assisted Networking Using Scheduled Transfer Protocol on Linux". He was a founding partner in Net People Oy, the first ISP in northern Finland (later merged with Nixu Oy) and Clarified Networks Oy. He is currently finishing up his PhD studies at the Oulu University Secure Programming Group (OUSPG).

Atte Kettunen, MSc, is a security researcher at the Oulu University Secure Programming Group (OUSPG). Since 2011 he has successfully fuzzed Firefox and Chromium and found dozens of vulnerabilities in them. Atte has quickly become one of the top reporters in both browsers' bug bounty programs.

Juha Röning is Professor of Embedded Systems in the Department of Computer Science and Engineering at the University of Oulu. He is principal investigator of the Biomimetics and Intelligent Systems Group (BISG). He is also leading Oulu University Secure Programming Group (OUSPG). There are two spin-off companies established from this research: Codenomicon and Clarified Networks. He has two patents and has published more than 300 papers in the areas of computer vision, robotics, intelligent signal analysis, and software security.