

demottja@msu.edu

June 1st, 2006

The Evolving Art of Fuzzing

By: Jared DeMott

1. Introduction

Fuzzing is a testing technique used to find bugs in software [26]. Often these bugs are security related since fuzzing is performed against the external or exposed interfaces of programs. Fuzzing is not used to establish completeness or correctness, the task of more traditional testing techniques. Instead, Fuzzing complements traditional testing to discover untested combinations of code and data by combining the power of randomness, protocol knowledge, and attack heuristics. Adding automatic protocol discovery, reading real-time tracer/debugger information, fault data logging, and multi-fuzzer sessions is the cutting edge in fuzzing tools.

Fuzzing is used by software companies and open source projects (e.g. Microsoft and Linux [2] [27] [29]) to improve the quality of their software, by vulnerability analysts to uncover and reports bugs (second party testing [3] [32] [28]), and by hackers to find and secretly exploit software [4]. Fuzzing is certainly not the only way these three parties explore the quality of software and there is plenty of debate about which methods are best. However, when compared with traditional software testing techniques or even source code audits and reverse engineering, fuzzing has been found effective and cost efficient [1].

Until recently, little was known about fuzzing; one company offers a fuzzing and reverse engineering class only to US government employees with a security clearance [6]. However, as fuzzing has matured, it has become an openly talked about and legitimate branch of software testing [1] [2]. There are even companies that produce and sell fuzzers, and that provide fuzzer development training as a sole or primary source of income [5] [6] [33] [40] [41]. This paper will explain the roots and current state of fuzzing, as well as discuss ongoing research and open questions.

2. Software Testing

a. Definition

The roots of fuzzing lie in testing. Software testing has been part of computing since the inception of computers and has been researched intensively. Software testing can be defined as:

“...the process used to help identify the correctness, completeness, security and quality of developed computer software.”[25]

However, there are limits to testing:

“...testing can never completely establish the correctness of arbitrary computer software. In computability theory, a field of computer science, an elegant mathematical proof concludes that it is impossible to solve the halting problem, the question of whether an arbitrary computer program will enter an infinite loop, or halt and produce output. In other words, testing is criticism or comparison that is comparing the actual value with expected one.” [25]

Later they expand their definition to include the open-ended nature of investigation which we will see relates to fuzzing:

“...effective testing of complex products is essentially a process of investigation, not merely a matter of creating and following rote procedure. ... Although most of the intellectual processes of testing are nearly identical to that of review or inspection, the word testing is connoted to mean the dynamic analysis of the product—putting the product through its paces.” [25]

Along with another definition we see the need for automated tools to help enable human creativity.

“Software testing can be defined as a process of executing [symbolically or dynamically] a program with the intent of finding errors. A successful test is one which uncovers an as yet undiscovered error. Hence, testing should be done with well-planned test cases, which ensure execution of all possible basic paths in the program. This is a time consuming task; software testing tools that can reduce the testing time without reducing the thoroughness are very much valuable in this context.” [16]

It is clear that the ability to deliver quality software requires more than a quick “test before ship”. A development process, with quality as a priority from the beginning (bug **prevention** as well as discovery), is needed. This process is called formal methods in software

engineering. Under the formal methods umbrella is software quality assurance. Under quality assurance we find software testing and its many branches [30].

b. Testing Types

Here are some of the testing techniques:

- **Black box testing** - not based on any knowledge of internal design or code. Tests are based on requirements and functionality.
- **White box testing** - based on knowledge of the internal logic of an application's code. Tests are based on coverage of code statements, branches, paths, conditions.
- **Unit testing** - the most 'micro' scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code. Not always easily done unless the application has a well-designed architecture with tight code; may require developing test driver modules or test harnesses.
- **Incremental integration testing** - continuous testing of an application as new functionality is added; requires that various aspects of an application's functionality be independent enough to work separately before all parts of the program are completed, or that test drivers be developed as needed; done by programmers or by testers.
- **Integration testing** - testing of combined parts of an application to determine if they function together correctly. The 'parts' can be code modules, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to client/server and distributed systems.
- **Functional testing** - black-box type testing geared to functional requirements of an application; this type of testing should be done by testers. This doesn't mean that the programmers shouldn't check that their code works before releasing it (which of course applies to any stage of testing.)
- **System testing** - black-box type testing that is based on overall requirements specifications; covers all combined parts of a system.
- **End-to-end testing** - similar to system testing; the 'macro' end of the test scale; involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.
- **Sanity testing or smoke testing** - typically an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is crashing systems every 5 minutes, bogging down

systems to a crawl, or corrupting databases, the software may not be in a 'sane' enough condition to warrant further testing in its current state.

- **Regression testing** - re-testing after fixes or modifications of the software or its environment. It can be difficult to determine how much re-testing is needed, especially near the end of the development cycle. Automated testing tools can be especially useful for this type of testing.
- **Acceptance testing** - final testing based on specifications of the end-user or customer, or based on use by end-users/customers over some limited period of time.
- **Load testing** - testing an application under heavy loads, such as testing of a web site under a range of loads to determine at what point the system's response time degrades or fails.
- **Stress testing** - term often used interchangeably with 'load' and 'performance' testing. Also used to describe such tests as system functional testing while under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database system, etc. Stress testing might also include unexpected environmental conditions such as a shortage of processing, memory, or disk resources.
- **Performance testing** - term often used interchangeably with 'stress' and 'load' testing. Ideally 'performance' testing (and any other 'type' of testing) is defined in requirements documentation or QA or Test Plans.
- **Usability testing** - testing for 'user-friendliness'. Clearly this is subjective, and will depend on the targeted end-user or customer. User interviews, surveys, video recording of user sessions, and other techniques can be used. Programmers and testers are usually not appropriate as usability testers.
- **Install/uninstall testing** - testing of full, partial, or upgrade install/uninstall processes.
- **Recovery testing** - testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.
- **Failover testing** - typically used interchangeably with 'recovery testing'
- **Security testing** - testing how well the system protects against unauthorized internal or external access, willful damage, etc; may require sophisticated testing techniques. A proper security analysis (not just running nessus) is still fairly rare.
- **Compatibility testing** - testing how well software performs in a particular hardware/software/operating system/network/etc. environment.
- **Exploratory testing** - often taken to mean a creative, informal software test that is not based on formal test plans or test cases; testers may be learning the software as they test it.

- **Ad-hoc testing** - similar to exploratory testing, but often taken to mean that the testers have significant understanding of the software before testing it.
- **Context-driven testing** - testing driven by an understanding of the environment, culture, and intended use of software. For example, the testing approach for life-critical medical equipment software would be completely different than that for a low-cost computer game.
- **User acceptance testing** - determining if software is satisfactory to an end-user or customer.
- **Comparison testing** - comparing software weaknesses and strengths to competing products.
- **Alpha testing** - testing of an application when development is nearing completion; minor design changes may still be made as a result of such testing. Typically done by end-users or others, not by programmers or testers.
- **Beta testing** - testing when development and testing are essentially completed and final bugs and problems need to be found before final release. Typically done by end-users or others, not by programmers or testers.
- **Mutation testing** - a method for determining if a set of test data or test cases is useful, by deliberately introducing various code changes ('bugs') and retesting with the original test data/cases to determine if the 'bugs' are detected. Proper implementation requires large computational resources." [30]

Fuzzing is a semi-automated process (human interaction to determine best uses and results are currently needed) that combines many of the above techniques: functional, black box, exploratory, stress, and security testing. Codenomicon uses the term "negative testing" or "robustness testing" to refer to fuzzing. While there are many definitions/terms we'll use the term fuzzing throughout this paper. The details of fuzzing will be examined in detail later in the paper.

Most companies know that producing and delivering patches to fix buggy software is more expensive than quality development from the beginning [27]. There are ratings (CMM levels) for how well a company creates software [23]. As with all such ratings the validity and practical meaning are constantly questioned [24]. Testing functionality and security is a very involved process. Some testers focus on ensuring all the code has been touched and works correctly, but omit security testing. That is why so many products are released with security vulnerabilities.

c. Better Development

Some companies, such as Microsoft, have developed their own formal and secure software development method (Security Development Lifecycle or SDL) for producing higher quality products [27]. Microsoft agrees that the cost-benefit ratio for adopting such a secure development process is favorable. In their development model we see many techniques such as secure design and implementation, multiple security testing techniques (including fuzzing), static analysis with source code auditing tools, and even manual security code reviews. This has sharply decreased the number of exploitable bugs externally reported in their products, but still bugs persist. 100% secure implementations are as impossible as 100% thorough testing. Even if all bugs currently known to be vulnerabilities could be removed, new classes of bugs may be discovered. However, it is clearly possible to do better. The quote below shows the current Microsoft stance on fuzzing:

“Heavy emphasis on fuzz testing is a relatively recent addition to the SDL, but results to date are very encouraging. Unlike the static code-scanning tools, fuzz-testing tools must be built (or at least configured) for each file format and/or network protocol to be tested; because of this, they are often able to find errors missed by static analysis tools. Threat models help product teams prioritize interfaces and formats for testing. The results of fuzz testing are not totally deterministic (the tools are run for a finite number of cycles and are not guaranteed to find every bug) but experience has shown that an affordable level of fuzz testing is likely to find “interesting” bugs that might otherwise be exploited as security vulnerabilities.” [27]

One method to ensure that products can and will be more thoroughly tested is adhering to a coding design that has high **testability**. If two code sets were compared and one was easy to test and the other was not, the former would be said to be better code [20]. Good internal data structures, instead of “tricky” pointer arithmetic, are one example. The data structures could be printed at any point in the code to determine the state of the data, while that might not be possible in the other case.

Most professional testers will tell you that automating testing via some type of test tool is a necessary aid, but that the tool should not replace creative thought in testing [17][18]. Most large companies are now using a combination of tools. Testing tools (could include a fuzzer) and code coverage suites, which are used to measure how well the testing went. Code coverage metrics are very useful, but can be abused [18]. This will be discussed in more detail later.

d. Summary

Testing is a very complex field. The focus of this paper will only be on a particular test tool known as a fuzzer. Standard software testing techniques focus on the correctness of software. Fuzzing concentrates on finding (exploitable) bugs. Both types of testing are important: if software does not do what it is supposed to do, that is a problem; if software has exploitable vulnerabilities that is a different problem. The advantage of looking for exploitable bugs is that one examines only part of a programs task and one focuses on breaking the code. Together that is a very different task than checking for correctness so it is not surprising that it can produce different results. Errors of omission are an interesting example. If a software writer omits some feature, say a proper bounds check, then it is not in the software to be checked and standard testing tools may miss the omission [18]. Fuzzers can find such errors.

3. Fuzzing

a. Definition

Fuzzing can be defined as:

“A highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. The name comes from modem applications’ tendency to fail due to random input caused by line noise on “fuzzy” telephone lines.” [1]

The first person credited with carrying out a rudimentary form of fuzzing was Barton Miller from the University of Wisconsin-Madison in 1990 [7].

“There is a rich body of research on program testing and verification. Our approach is not a substitute for a formal verification or testing procedures, but rather an *inexpensive* mechanism to identify bugs and increase overall system reliability. We are using a coarse notion of correctness in our study. A program is detected as faulty only if it crashes or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures.

While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states.” [7]

He found that supplying totally random input would crash many of the core UNIX utilities. Crashes indicate abnormal program behavior which in turn indicates potentially exploitable vulnerabilities. Programming, testing, flaw detection, and quality assurance have greatly improved since then, but code has also become more complex and bugs still exist. Fuzzers that supply totally random characters may yield some fruit, but in general, won't find many bugs these days. As such, it is necessary for fuzzers to become more complex (evolve), if they hope to uncover such **buried** or hard to find bugs. Very obscure bugs have been called **Second Generation Bugs**. They might involve things like multi-path vulnerabilities such as non-initialized stack or heap bugs [36].

b. Terms and Examples

Coverage is an important term that is used in testing. Coverage typically refers to code coverage; that is, how much of the existing source or compiled code has been tested [1]. Coverage could also point to path, data, or a variety of other coverage metrics. A related term is **attack surface**: the amount of code actually exposed to an attacker [1]. Some code is internal and cannot be influenced by external data. This code should be tested, but cannot be externally fuzzed and since it cannot be influenced by external data it is of little interest when finding vulnerabilities. Thus, our interests lie in *coverage of the attack surface*. This is especially true for security researchers. Quality assurance professionals may be tasked to fuzz all of the code.

"A **trust boundary** is any place that data or execution goes from one trust level to another, where a trust level is a set of permissions to resources." [1] For example, data sent across a network to an application that **parses** that data is an example of crossing a trust boundary. If the root user of a UNIX system is the only one able to start a given application (via command line arguments), priority would probably go to fuzzing the network interface instead of the command line arguments for two reasons: a remotely exploitable bug is more interesting to attackers (since it can be done remotely), but in terms of trust boundaries an elevation of privilege (from none to whatever the process runs as) can occur in the remote situation. Conversely, if the user must already be root to gain root (unless a tricky method is devised to run the binary without root privileges) nothing has been gained, plus the attack would only be local (less of a liability to companies). Priority is important to software

companies and attackers alike because the problem of finding bugs is difficult and time consuming. Neither is willing to waste much time (money) for purely academic reasons; fuzzing is known for its ability to produce results. The word "parses" is in bold above because complex parsers often have bugs. Thus, this a good place to fuzz. Fuzzing tends to use past mistakes as a guide to finding future bugs (source code auditors and reverse engineers will also leverage knowledge of past bugs).

Input source and **input space** are similar terms that refer to how data will be generated (and ultimately delivered) to the application to be fuzzed (**target**). The input space is the entire set of all possible permutations that could be sent to the target. This space is infinite, and that is why heuristics are used to limit this space. **Attack heuristics** are known techniques for finding bugs in applications. The examples below illustrate a few simple heuristics against an uncomplicated string-based client-server protocol:

Buffer overflow:

```
[Client]-> "user jared\r\n"
           "user Ok. Provide pass.\r\n" <-[Server]
[Client]-> "pass <5000 'A's>\r\n"
```

Integer overflows: (zero, small, large, negative: wrapping at numerical boundaries – 2^4 , 2^8 , 2^{16} , 2^{24} : wrong number system – floats vs. integers)

```
[Client]-> "user jared\r\n"
           "user Ok. Provide pass.\r\n" <-[Server]
[Client]-> "pass jared\r\n"
           "pass Ok. Logged in. Proceed with next command.\r\n" <-[Server]
[Client]-> "get [-100000.98] files: *\r\n"
```

Format attacks:

```
[Client]-> "user <10 '%n's>"
```

Parse error: (NULL after string instead of `\r\n`)

```
[Client]-> "user jared0x00"
```

Parse error: (Incorrect order and combined command in one pkt)

[Client]-> "pass jared\r\nuser jared\r\n"

Parse error: (Totally random binary data)

[Client]-> "\xff\xfe\x00\x01\x42\xb5..."

Parse error: (Sending commands that don't make sense – multiple login.)

[Client]-> "user jared\r\n"

"user Ok. Provide pass.\r\n" <-[Server]

[Client]-> "pass jared\r\n"

"pass Ok. Logged in. Proceed with next command.\r\n" <-[Server]

[Client]-> "user jared\r\n"

Parse error: (Wrong number of statement helpers such as `./`, `{`, `(`, `[`, etc.)

[Client]-> "user jared\r\n"

"user Ok. Provide pass.\r\n" <-[Server]

[Client]-> "pass jared\r\n"

"pass Ok. Logged in. Proceed with next command.\r\n" <-[Server]

[Client]-> "get [1] files: {{../../../etc/password\r\n"

Parse error: (Timing issue with incomplete command termination)

[Client]-> "user jared\r" (@ 10000 pkts/second with no read for server response)

Again, it is obvious that the input space is infinite. It would technically be possible to send totally random data and get the same effect as using heuristics, but instead of the fuzzer runtime being finite and reasonable it would be infinite and therefore unreasonable.

Furthermore, with an increased number of tests comes an increased load of logging. It is desirable to be able to detect when a fault (bug) has occurred (been found); that's the whole reason we're fuzzing. Detection can be accomplished either by logging and time stamping everything generated by the fuzzer and comparing to application logs, and/or by incorporating a debugger which can detect and log signals (like SEGV) received by the application(s) under test. So the goal is to cover every **meaningful** test case (without too much duplication or unneeded sessions) and to log the ones that succeed in causing the target to fail in some way.

“Other ways to check application correctness during fuzzing include looking for spikes in the program’s memory usage or CPU utilization, which could indicate that the application is using the malformed input to make calculations for memory allocation. It could also indicate an integer overrun condition if the program performs arithmetic operations on the input values.”

[1] Tools such as OllyDbg, GDB, tools by sysinternals, etc. will be useful here.

c. Fuzzer Types

Fuzzer test cases are generated by the input source and attack heuristics. Test cases could also be developed manually instead of automatically, though it is not the preferred approach since many test cases will be needed. Input source refers to the type of fuzzer being used. There are two main types: **Generation** and **mutation** [1]. These have also been called full-blown and capture-replay. A generation fuzzer is a self-contained program that generates its own semi-valid sessions. (Semi-**invalid** has the same meaning in this context.) A mutation fuzzer takes a known good session, file, etc. and mutates the capture before replay to create semi-valid sessions. Mutation fuzzers are typically **generic fuzzers** or **general purpose fuzzers**, while generation fuzzers tend to be specific to a particular protocol or application. Understanding the nature of the attack heuristics, how this mutation or generation is accomplished, is important.

d. How Fuzzers Work

There are four main ways this semi-valid data can be created: **Test cases, cyclic, random, or library.**

- **Test cases** refers to a fuzzer that has X numbers of tests against a particular protocol. This type of fuzzer will only run against the protocol it was created to fuzz. This type of fuzzer will send the same tests each time it is run and typically keeps a good record of each test. This type of test tool most closely resembles traditional automated test tools or stress testing tools. The number of tests is relatively small compared to more random methods, but each test has a specific purpose. Typically these are generation fuzzers.
- Another way to send semi-valid input is to **cycle** through a protocol by inserting some type of data. For example, if we want to search for buffer overflows in the user name we could cycle data sizes from 1-10000 by say 10:
[Client]-> "user ja<1 A>red\r\n"

```
[Client]-> "user ja<11 A's>red\r\n"
```

```
[Client]-> "user ja<21 A's>red\r\n"
```

...

This method yields a deterministic number of runs and thus a deterministic runtime. (One might argue that only fixed buffers around known boundaries should be tested, but off-by-one errors (if a static buffer is of size 100 bytes sending exactly 101 bytes would be needed to trigger the bug) are a known issue.)

- One could also choose to keep **randomly** inserting data for a specified period of time:

```
[Client]-> "user ja<10000 A's>red\r\n"
```

```
[Client]-> "user ja<12 A's>red\r\n"
```

```
[Client]-> "user ja<1342 A's>red\r\n"
```

...

Random fuzzers can be repeatable if **seeded** by the user. This is important for regression testing.

- **Library** refers to a list of known useful attacks. Each variable to be tested should be fuzzed with each type of attack. The order, priority, and pairing of this search could be deterministic, random, or weighted. Mature fuzzers will typically combine many of the above techniques.

e. Intelligent Fuzzing

Fuzzers can become as fancy as the imagination will allow. This is sometimes called the **intelligence** (protocol knowledge) of a fuzzer; **intelligent fuzzing**. Consider a fuzzer that randomizes the test type, position, and protocol leg in which to place the attack:

```
[Client]-> "us<50000 \xff's>er jaed\r\n"
```

```
-----loop 1-----
```

```
[Client]-> "user ja<12 %n's>red\r\n"
```

```
    "user Ok. Provide pass.\r\n" <-[Server]
```

```
[Client]-> "\x34\x56\x12\x...\r\n"
```

```
-----loop 2-----
```

```
[Client]-> "user ja<1342 \x00's>red\r\n"
```

```

-----loop 3-----
[Client]-> "user jared\r\n"
        "user Ok. Provide pass.\r\n" <-[Server]
[Client]-> "\x04\x98\xbb\x...\r\n"
-----loop 4-----
...

```

We note that valid data, such as "user jared", is transformed into semi-valid data, such as "usAAAAAAAAAAAAAAAAAAAAAAer jared". The insertion could be done by working from a capture file or from internal generation of the protocol (or possibly a combination of the two). As Mr. Oehlert mentions, intelligent vs. unintelligent is always a trade off [1]. Creating intelligence takes more work, and typically begins to assume things about the protocol. For example, suppose our intelligent fuzzer assumes that the *user* command must always be correct and the first command, therefore there is no need to test it. Well, 98% of the time that's a true assumption. But what about the odd case where a particular implementation reads in an arbitrary amount of data until the first space character? Or what if a command prior to *user* hangs the internal state machine? A balanced approach seems best: intelligence enough to decrease the runtime, but not so much that it weakens the fuzzer (i.e. makes the same poor assumptions the programmer did), and possibly costs too much to produce.

Another useful extension of intelligent fuzzing is for protocols that require calculations to be made on data that is sent or received. For example, an Internet Key Exchange (IKE - IPsec key exchange protocol) fuzzer would require the ability to deal with the cryptography of IKE if the fuzzer ever hopes to advance into the IKE protocol. Because of the complex nature of IKE, capture-replay is made very difficult. IKE might be a good candidate for a generation fuzzer with intelligence.

Still another intelligent feature that could be built into fuzzers is known as **pattern matching** [1] [42]. In the above heuristic examples, a pattern matching fuzzer could automatically find the strings in a capture file and build attacks based on that. It could also automatically find and fix length fields if the fuzzer inserts data [14]. **Boundary** conditions should be noted as important and included in any good pattern matching library. "If a given function *F* is implemented in a program and the function has two parameters *x* and *y*, these two have known or unknown boundaries $a < x < b$ and $c < y < d$. What boundary testing does is test the function *F* with values of *x* and *y* close to or equal to *a*, *b*, *c* and *d*." [21] Thus, if our

intelligent fuzzer finds a variable with the value 3, it might first exercise the program under test with the values 2 and 4. Trying other key numeric values such as zero, negative, large positive, etc. might be the next phase. Finally wilder data such as wrong numeric systems, non-numeric, special characters, etc. would complete a sampling of each data quadrant without trying every possible value. Does this approach theoretically guarantee the discovery of the magic value (if one exists) that exercises a bug? No, but this is an example of how attack heuristics balance runtime and effectiveness.

f. Additional Factors that Influence Fuzzers

Finally, there are a few heuristics for fuzzing that don't directly relate to the data. In fact they don't directly relate to fuzzing, but more closely coincide with the umbrella term **vulnerability analysis** or **vulnerability research**. A vulnerability analyst (VA), security researcher (SR), or vulnerability researcher (VR) is one whose job is to search for bugs using any techniques available. Reverse engineering (RE), source code audits, and fuzzing are the three main techniques. The former two are typically static analysis and the later is an active search and any combination in between can and should be used. Vulnerability analysts tend to leverage the history of the application under test. Mr. Moore talks about SBDA ("same bug, different app") in a presentation recently delivered [10]. The idea is that coders sometimes make the same mistakes. Pretend 50% of web applications have had a particular bug in the GET command. Fuzzing the GET command in your new web application would be a good place to start looking for bugs. Historical weaknesses factor into fuzzing prioritization and attack heuristic creation.

Along those same lines is the localization or similarity search for bugs. This involves more intuition from a VA. If a portion of the target code has a minor (non-exploitable) bug, it might be wise to zero in on this area and fuzz more extensively. This is when the VA is like a shark and smells blood in the water. Generic fuzzing might suddenly switch to small targeted scripts, based on RE or source code work. Or on a different note, when format bugs were first announced everyone should have quickly retested all their code as this heuristic was not previously known. (Though in theory, this wouldn't have taken too long to discover with some randomness in attack creation.) So, weak areas of an application or new heuristics can also influence fuzz priority.

g. Error Handling Routines

There has been a substantial amount of work done to fuzz the exception handling routines of programs. One such company, Security Innovation, Inc., has a tool called Holodeck whose primary function is just that [15]. The idea is that while you are fuzzing, the server may never run out of open files, hard disk space, memory, etc. Holodeck can simulate those error conditions (they call it **fault simulation**), and allow for the error-handling routines to be systematically tested.

h. Real Life Fuzzers

Much work has been done on generation fuzzers. Specifically a group known as Oulu has created a tool called the Mini-Simulation Toolkit (PROTOS) [34]. They used a context-free grammar to represent the protocol (language) and build anomalous test cases. This work is now carried on by Codenomicon, Ltd. Their work is impressive at finding bugs and creating a baseline tool: specific implementations of a protocol that are found to have flaws *don't pass*, and those that are bug free *pass* the test. They admit that this approach is likely to have a pesticide effect: wide spread use of the tool will become less effective as the tool is run and bugs are repaired; shared knowledge (code and techniques) becomes more and more immune to the tool. This should happen to every bug finding tool that doesn't employ randomness or isn't updated; compare to regression testing.

In 2005, Mr. DeMott wrote a generic fuzzer (among other tools) known as GPF (General Purpose Fuzzer) [42]. GPF employs many of the elements discussed in this paper. It will be publicly released (along with this paper, presentation slides, an IKE fuzzer tool, and a local *NIX command line fuzzing tool) on August 5th at DEF CON 14 [43]. Go to www.appliedsec.com for more information.

Impressive work on a generic fuzzing technique and tool (known as Autodafe) has recently been done by Mr. Vuagnoux [22]. Autodafe includes automatic protocol detection with manual updating available, a large list of attack stings, and an incorporated debugger to dynamically place weights on areas of target code that utilize external data in dangerous functions. Multiplying the number of attack strings by the number of variables to fuzz yields the complexity. By minimizing and ordering the tests, the overall runtime is decreased.

This is certainly efficient when compared with other more random methods. However, the power of randomness is completely replaced by lists of known issues. Much is unknown

about the exact workings of Autodafe because the tool has yet to be released as of the writing of this paper.

There are many other fuzzers for sale or download [41].

i. Taxonomy

There has been little research comparing and contrasting fuzzers types. Each seems to have different strengths and weaknesses. Generation fuzzers have the advantage of possibly being more complete and having test cases tailored to the particular protocol. It's difficult for generic fuzzers to get a good capture that includes every possible leg/state of a protocol. But good generic fuzzers can easily be extended to be as complete as generation types [22] [42]. Generation works against only one protocol, and the fuzzer must be re-written or extended for any new fuzzing task. Generic fuzzers need a new capture or plug-in for each new protocol, but the meat is already there and this core knowledge (timing, heuristics, attack list, etc.) is retained and extended across projects. With generic fuzzers, security researchers have a consistent interface for fuzzing. Generation fuzzers typically require more programming per task, but the test results are sometimes better recorded (since there tends to be less tests). Generic fuzzers require more up front work, but tend to produce more tests, and the results can also be quite detailed. In either case, developing fuzzers that are specific to a particular source code tree is expensive. Fuzzers tend to be developed for particular protocols and used on multiple projects to increase their usefulness and decrease their cost. That is when a company that writes and FTP server may just purchase an FTP fuzzer. Is this the best option? The answer lies in the organizations in-house expertise and budget. Or if that same company had or could buy a generic fuzzer a FTP capture would be about the only thing needed. (Besides the obvious of an experienced test team familiar with FTP and VA work in general.)

The fuzzer that finds more bugs is ultimately the best, but this is very difficult to measure since before testing it is unknown if bugs exist. If we already know bugs exist the fuzzer could be tailored to find them. An experiment where independent parties develop a fuzzer (one generation and the other generic) for a protocol which has bugs that only a third party knows about would shed some empirical light. Yet, more theory would be better. It's safe to assume that all fuzzers could miss some bugs. Every situation, protocol, and/or application is different. The best testing method(s) seems to vary from project to project and should be determined by the projects leaders and an **experienced** testing team.

j. Vulnerability scanners and fuzzers

A vulnerability scanner sweeps a host or network for previously discovered bugs via a list of vulnerability signatures; compare to traditional virus cleaning software. However, many virus products are doing more to protect your computer these days. Many virus companies have combined personal firewalls, web privacy, spam filters, anomalous system behavior detection and reporting, etc. Thus, it seems likely that professional fuzzing or vulnerability scanning companies will combine fuzzing and scanning technologies in new and interesting ways.

k. Summary

As each new research project is brought to life and grown, the balance beam flips back and forth. For example, more random tools and scripts (like Miller's original tool) were at one point state of the art. Fuzzing frameworks like SPIKE came out and showed us all a better way to fuzz. PROTOS hit the scene and was clearly superior. Many impressive generic fuzzers like GPF and Autodafe are in use as I write. Like so many fields, fuzzing is evolving.

Another interesting reason fuzzing has been effective is that it fills a void or **gap**. Testing is always incomplete in some way. Therefore, if an external tool does something different from internal company tools, even if it is an inferior tool, it may produce different results (i.e. find a bug they missed). Therefore, private fuzzers may always retain some value. Companies are beginning to fuzz more and more, and some have turned to commercial fuzzers. Considering all this, it makes one ponder, will hackers eventually need new ways to find bugs if fuzzers became very effective and continue to be widely adopted by companies? Companies will/are producing more secure code and running them in safer (stack/heap protected) environments. This is good news for security, but there will always be some testing gap and so it seems likely there will be a use for tools like fuzzers (and all other VA work) for the foreseeable future.

4. Conclusion

Bug detection tools known as fuzzers are a useful part of software testing and vulnerability analysis. The best fuzzers of today are built by experienced vulnerability analysts and testers, and employ the power of automatic protocol descriptions, randomness, known useful heuristics, tracing/debugging, logging, and more.

5. Open Questions and Future Work

a. Formality

You will have noticed no formalism of fuzzing in this paper: we have not been able to find any formalism related to fuzzing. It is clearly a difficult problem because *vulnerability* is not a well-defined concept except the implicit notion that some error has occurred and might be exploited. Formalizing *mistakes* is difficult. Traditional testing can focus on what is expected in the design whereas fuzzing focuses on the unexpected. If you could formalize the unexpected, it might be expected. Exposing some formalism would be wonderful, and we will try, but we will be surprised if we succeed.

b. Metrics

Formal fuzzing metrics do not exist, yet metrics would be extremely useful. Two fundamental and practical questions related to metrics exist:

- what makes one fuzzer “better” than another and
- how long should it run (beyond “a while” or “until we find a bug”)?

Many people have created fuzzers for many protocols and applications, and it’s clear that some fuzzers are better than others at finding bugs. Why? Factors involved are: Completeness, complexity, incorporated heuristics, experience of the fuzzer developer and test team, familiarity with the protocol, type of fuzzer, type of protocol/application, length of the run(s), total testing time, amount of previous in-house company testing, budget, etc.

The quality of a fuzzer is related in some way to coverage—code which has been executed as part of a test was “covered.” Certainly, if exposed code is **not** covered, the code was not tested, and this is not good. However, simply covering all code at least once does not define a complete test. We have come to the conclusion that the current definition(s) of coverage are insufficient [18]. It is clear that although all of the desired code could be covered; it may not have been covered “right”. *Particular* paths, with *particular* inputs, through the code could lead to bugs in code that has already been covered. Again, coverage can only tell us if we’ve missed code, not if we’ve correctly covered code or if more code should exist (**faults of omission**). Thus, we need knowledge of *all* paths with *all* data to claim something more useful about coverage. This is the infinite problem.

As a start, it could be useful to combine code coverage with bounds on the inputs which test that code. For example, we wonder if there is some new measure, something like an

“iterations per state” count, which could be used as a finite measure of state coverage instead of code or path coverage. With that in hand we might be able to estimate how much useful work a fuzzer has done. If that measure was effective, we would know something about the quality of the fuzzer.

We would also like to know how long this new fuzzer must run for it to be called a “good” (complete, sufficient, etc.) run. If a fuzzer is deterministic we will easily know how long it will run. But if heavy amounts of random or dynamic decisions are employed it becomes a more difficult problem. If we are able to produce a good test (both in terms of coverage and time) of a given protocol, we believe those that fuzz would highly benefit from this work.

c. Generic Fuzzer

To date, most fuzzers have been custom built for each application or protocol. Fuzzers which can be readily tuned for particular protocols exist. For example, PROTOS takes as input a BNF specification of a protocol which it uses to generate test cases. However, constructing the needed BNF file is difficult. We have been looking into developing a generic fuzzer which could learn its target and generate test cases appropriately.

We intend to build on existing fuzzing knowledge to advance the state of fuzzing. For example: if after Autodafe completes, and no interesting bugs were discovered, it would be advisable to continue fuzzing with increased randomness. Random attacks (data mutations), random states, random number of concurrent sessions, etc. will broaden the input space. As no bugs continue to be discovered it’s likely the runtime will approach the input space – infinite. A reasonable cutoff time should be able to be determined to call the run complete. Or perhaps a tool with some known cut off time (say we have a maximum of 20 days to fuzz) could use that knowledge to order test cases appropriately.

Additionally a generic fuzzer with all of the following attributes would be novel and very useful (GPF and Autodafe contain much but not all of this):

- Automatic Protocol Detection

Our generic fuzzer should listen and learn. That is, our source data comes from sniffing a live session. The generic fuzzer should then convert that data into a neutral format where it can be manually extended by the tester. Plug-ins can be developed to allow calculations based on received data, such as encrypted protocols. If done right, this technique will allow for the rapid fuzzing of most any text or binary protocol.

- Tokenizing

The generic fuzzer should then tokenize each piece of data to an internal format. Stings, binary data, and length fields should be automatically detected and fuzzed upon replay. This makes the process from capture to fuzz that much faster and useful.

- Strong Attack Heuristics

The capture data should then be sent back to the target with intelligent and/or random attacks intelligently and/or randomly inserted. Null, format, or large strings are examples of intelligent replays. Changing the post-amble (`\r\n` for string protocols), length fields, parsers (brackets, parenthesis, quotes, etc.) are further examples. Bits could be randomly flipped and/or inserted in specific and/or random positions. Another useful technique is to change or randomize the ordering and paring in which valid commands are sent to the target.

- Intelligent Randomness

Properly weighting what happens (i.e. which heuristics and how) at each stage will influence how well our fuzzer performs, particularly in terms of time. There has been no research other than Vuagnoux's suggesting a proper method for doing this. Much of what's done is based on the notion of proper functionality a "majority" of the time (i.e. semi-invalid...not totally invalid). Or it's simply based on personal preference or "gut instinct".

- Debugging

The generic fuzzer should be remotely attached to the process under test to determine when a fault occurs. Also the debugger could be used to get real-time statistics on code/path coverage. Mr. Vuagnoux uses the debugger to dynamically identify dangerous functions and weight the fuzzer toward those paths. Vuagnoux says he's solved another major hurdle by creating a tracer that runs on both Windows and UNIX so his tool is applicable for either platform. There is one minor issue with attaching a debugger to a process to determine when faults have occurred. In rare cases, attaching a debugger will actually change the program enough that it will mask the presence of certain timing related bugs. The latest *sendmail* bug is the best example in a while.

- Distributed Fuzzing

It is possible to split the load of fuzzing across hosts (targets = "fuzzies" and sources = "fuzzers"). This should decrease the total runtime. Tasking in a deterministic fashion will require a central fuzz server that partitions out the work load. Another possible method to

decrease run time is to start multiple instances running with randomness as the driving force. For generic fuzzers, the capture file could even be file fuzzed before each instance is started, using a list of valid protocol commands and known useful heuristics. [42] The problem with this approach is that debugging is difficult. A central debug relay would be required. It's also difficult to determine a sane stopping point, other than "when we find a bug". Imagine if a crash happens when 50 fuzzers are running as fast as possible and no central debug relay is in place; for tricky bugs it could be difficult to say why/when/if the application died. On the other hand this technique may be useful for targets where no bugs can be found with any other technique. The target will be under heavy load, handling many malformed sessions as fast as possible. And again, the amount of test cases covered could be dramatically increased.

d. Evolving Fuzzer

What about a fuzzer which evolves? Genetic algorithms (GAs) are good at solving problems for which no optimal solution exists, and they have been used with limited success to test software [37, 38]. Given those results, it may be possible to formulate an expression for gene as well as an objective function for a fuzzer to which we could apply GA techniques. In addition, genetic programming might be applicable to develop code which evolves [39]. Such an approach might not only result in a generic fuzzer, but it might also result in a fuzzer which can learn, grow, and evolve. There are many emerging methods for dynamically determining if a program has violated (a bug) it's original construction [35]. Combining that technology into an adaptive fuzzer could prove to be very interesting.

Works Cited

- [1] Peter Oehlert, "Violating Assumptions with Fuzzing", *IEEE Security & Privacy*, Pgs 58-62, March/April 2005
- [2] Michael Howard, "Inside the Windows Security Push", *IEEE Security & Privacy*, Pgs 57-61, January/February 2003.
- [3] Dave Aitel, <http://www.immunitysec.com/company-people.shtml>, Developer of SPIKE a fuzzing framework and founder of Immunitysec a security research company.
- [4] Kevin Mitnick, <http://www.tectonic.co.za/view.php?src=rss&id=839>, Infamous hacker speaks on using fuzzing to hack.
- [5] Gleg Ltd., <http://www.gleg.net/>, A company that sells fuzzers.
- [6] Liveammo. http://www.liveammo.com/LiveAmmo_Reverse_Engineering_Training.php Offers various high tech training courses including a "restricted" one on RE and Fuzzing.
- [7] Barton P. Miller, Lars Fredriksen, Bryan So, "An empirical study of the reliability of Unix Utilities", *Communications of the ACM*, 33(12):32--44, December 1990.
- [8] M. Suttan and A. Greene, "File Fuzzing", *BlackHat Conference*, Las Vegas, NV, July 2005
- [9] Ejovi Nuwere & Mikko Varpiola, "The Art of Sip Fuzzing and Vulnerabilities found in SIP", *BlackHat Conference*, Las Vegas, NV, July 2005
- [10] Brett Moore, "SBDA - same bug, different app", *Ruxcon 2005*, security-assessment.com
- [11] Giovanni Vigna, "Testing and Analysis", University of California Santa Barbara, <http://www.cs.ucsb.edu/~vigna>, Fall 2005
- [12] Ilya van Sprundel, "Fuzzing: Breaking Software in an Automated fashion", Decmber 8th, 2005, http://events.ccc.de/congress/2005/fahrplan/attachments/582-paper_fuzzing.pdf
- [13] Ilya van Sprundel, "Fuzzing", *22nd Chaos Communication Congress*, Berlin, 2005, <http://www.digitaldwarf.be/library.html>

- [14] Dave Aitel, "The Advantages of Block-Based Protocol Analysis for Security Testing", February, 4 2002. New York, New York.
http://www.immunitysec.com/downloads/advantages_of_block_based_analysis.txt
- [15] A Professional Software Testing Company,
<http://www.securityinnovation.com/holodeck/index.shtml>
- [16] Ghulam Khan, "Test Coverage Analysis: A method for Generic Reporting" Thesis for Master of Technology from Indian Institute of Technology, Kanpur. April 2000.
- [17] <http://www.io.com/~wazmo/qa/>, "A case for automated testing", Software Testing Hotlist - Resources for professional testers.
- [18] Brian Marick, "How to Misuse Code Coverage",
<http://www.testing.com/writings/coverage.pdf>
- [19] James A. Whittaker, "Stochastic Software Testing", *Annals of Software Engineering*, 4, 115-131, 1997
- [20] James A. Whittaker, "What is Software Testing? And Why is it so Hard?", *IEEE Software*, Pgs 70-79, Jan/Feb 2000
- [21] Felix 'FX' Lindner speaks out on Fuzzing. "The big Fuzz", *The Sabre Lablog*,
<http://www.sabre-labs.com/>, March 13, 2006
- [22] Martin Vuagnoux, "Autodafé: an Act of Software Torture", *22nd Chaos Communication Congress*, Dec. 2005, (<http://events.ccc.de/congress/2005/fahrplan/events/606.en.html>).
<http://autodafe.sourceforge.net>
- [23] CMM Basics. <http://www.karkhanisgroup.com/cmm/>
- [24] Christopher Koch, "Bursting the CMM Hype", CIO Magazine, March 2004
<http://www.cio.com/archive/030104/cmm.html>
- [25] Definition of software testing. http://en.wikipedia.org/wiki/Software_testing

- [26] Definition of fuzzing. <http://en.wikipedia.org/wiki/Fuzzing>
- [27] Steve Lipner and Michael Howard, "The Trustworthy Computing Security Development Lifecycle", March 2005, <http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dnsecure/html/sdl.asp>
- [28] The Digital Dwarf Society, a group that strongly believes in fuzzing.
<http://www.digitaldwarf.be/>
- [29] Scott A. Maxwell, "The Bulletproof Penguin", Secure Linux through fuzzing,
<http://home.pacbell.net/s-max/scott/bulletproof-penguin.html>
- [30] Rick Hower, "What types of testing should be considered",
http://www.softwareqatest.com/qatfaq1.html#FAQ1_1
- [31] Good page on various Fuzzing tools and techniques out there.
<http://www.scadasec.net/secwiki/FuzzingTools>
- [32] Roger Grimes, "The buzz about fuzzers",
http://www.infoworld.com/article/05/09/09/37OPsecadvise_1.html
- [33] Larry Seltzer, "The Future of Security Gets Fuzzy",
<http://www.eweek.com/article2/0%2C1895%2C1914332%2C00.asp>
- [34] Rauli Kaksonen, "A Functional Method for Assessing Protocol Implementation Security",
Technical Research Centre of Finland, VTT Publications
<http://www.ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/>
- [35] J. Giffin, S. Jha, and B. Miller, "Efficient context-sensitive intrusion detection", In 11th Annual Network and Distributed Systems Security Symposium (NDSS), San Diego, CA, Feb 2004.
- [36] Halvar Flake, "Attacks on Uninitialized Local Variables", Sabre-security.com, Black Hat Federal, 2006

- [37] A. Watkins, D. Berndt, K. Aebischer, J. Fisher, L. Johnson, "Breeding Software Test Cases for Complex Systems," *hicss*, p. 90303c, Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9, 2004
- [38] C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton, "Genetic algorithms for dynamic test data generation", Technical Report RSTR-003-97-11, RST Corporation, Sterling, VA, May 1997
- [39] Sara Reese Hedberg, "Evolutionary Computing: The Rise of Electronic Breeding," *IEEE Intelligent Systems*, vol. 20, no. 6, pp. 12-15, Nov/Dec, 2005
- [40] Codenomicon Ltd., <http://www.codenomicon.com/>, A company that sells fuzzers.
- [41] Matthew Franz, <http://www.threatmind.net/secwiki/FuzzingTools>, List of Fuzzers
- [42] Jared DeMott, <http://www.appliedsec.com>, Vulnerability researcher focused on Fuzzing, Released a free General Purpose Fuzzer (GPF) useful for finding bugs in most any protocol.
- [43] www.defcon.org, Unique Security Conference