Scalable Fuzzing in the Cloud for Software Security and Quality Assurance

Fred Concklin

Division 3 Thesis

Hampshire College, 2013

Abstract

EC2fuzz is a proof of concept tool for automating scalable cloud fuzzing with virtually no prior configuration. The tool is ultimately a Development Operations (DevOps) tool for building a customized cloud configuration tailored to fuzzing and executing fuzzing runs on the configured infrastructure. It performs both UNIX and Windows automation tasks, and performs automation tasks for Windows from a UNIX environment and implements UNIX network protocols in a Windows environment. EC2fuzz is able to take a stock Windows Server virtual machine and configure it to run additional binaries and programmatically install dependencies. It tunnels firewalled network protocols through SSH to allow for easy access to remote instances on the local filesystem, which makes installing files and collecting logs very simple. EC2fuzz represents a researched solution to difficult cloud automation tasks in multi-platform environments.

"***Lead in to the distributed stuff [...] Vision of how this will work [...] Its really easy to get excited here [...] We see a skynet-style fuzzing farm operating thousands of nodes from outer space scaling at will, autonomously based on a doctorate level heuristic with the ability to alert the devs when a serious issue is found.***

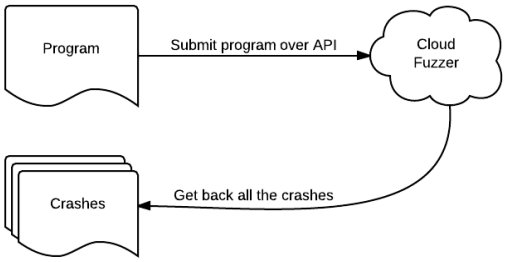
Hold on[...]"

- Peter Morgan, John Villamil, Matasano Security, DeepSec 2012 [1]

As my undergraduate thesis I wanted to work in “automated exploit pipelines.” This is a topic I had always been curious about but had never actually worked in or on. When I got to work researching I came across a paper called “Towards an automatic exploit pipeline” by Jared DeMott, et. al [2]. In this paper, DeMott provides a high level conceptualization of what he thinks an automated exploit pipeline would entail. From there, he addresses his own personal work in the first stage of his proposed exploit automation scheme. In this paper, DeMott describes a tool he has designed named Cluserfuzz. He describes the tool as having the following properties:

1. “distributed fuzzer … powerful enough to create numerous bugs from real world applications”
2. “prioritizing fuzzer output based on severity”
3. “GUI […] allows [...] analysts to steer the process at a few critical points”

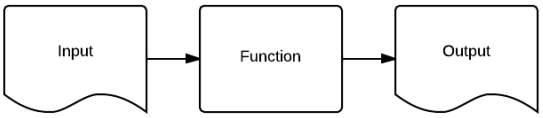
My Division 3 thesis basically resides within the realm of what else can be done in this first stage of the automation pipeline. I wanted to understanding fuzzing and how it could be automated. I started with the goal of trying to build a system which takes a whole bunch of fuzzers and then runs them, in tandem, on a computer program to try and find bugs. I wanted an API to submit a program and get back crashes without worrying users about the specifics. I had multiple goals. The first was to design a relatively large system that utilized cloud components. I would not only be learning how to design a system for fuzzing, but learning about application scalability and the creation of entire



virtual operating systems on the fly. This was very appealing to me and still is. However, it probably makes sense to build a working set of terms so everybody is on the same page.

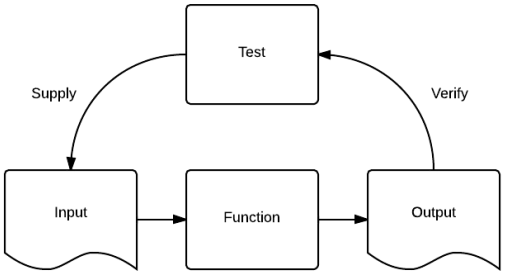
Fuzzing. What is it?

Fuzzing is one of the technologies at the core of this Division 3. Ironically, while it is a subfield which falls within Information Security and Quality Assurance, the overall description is somewhat anticlimactic. If I were to “fuzz” a computer program, I would be purposely modifying an input to the program in order to see how it responds to modified input. When you write software, you want to know that it works.



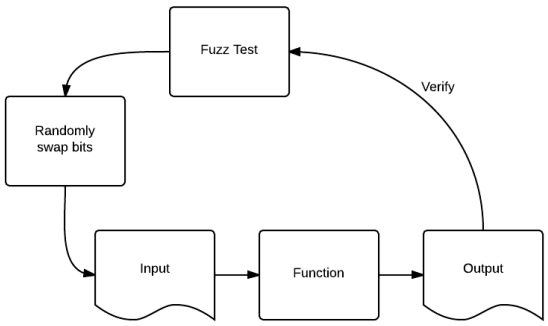
Oftentimes, when writing software, there are multiple combinations of pieces and/or multiple types of input a component is expected to receive. In order to make sure that newly introduced code or modifications in code haven’t changed the actual operation of the software, programmers write what are known as tests. The concept of a test here should be somewhat self-explanatory: you measure the performance of a piece of code against an expected result. To give an example, let’s say I write a simple function called “doubler” which takes an input, multiplies it by 2, and returns the output. Now, a test of this function would be verifying that it produces the expected results when given a known input. For instance, a test of doubler could be a simple wrapper which verifies that if doubler is provided 4 as an input, the output is 8.

Part of testing is making sure that you handle the unintended cases as well. For instance, what happens if doubler operates only on a datatype of int and is given a string? If there is no case logic for this scenario, the software may crash. As software gets more complex, so does the interaction of various software components. If



you don’t have good tests, you are probably likely to have more bugs in your code than if you have good tests. So let’s think about how fuzzing can be useful in the context of testing. If testing consists of supplying inputs to programs to see if they return an expected result, fuzzing can be considered a supplied input generation technique. The next question worth asking is what types of inputs does fuzzing generate and why are the useful? Fuzzing can ultimately generate a number of different inputs that can be useful in the context of software testing.

Stochastic Input Generation



This technique really gets at the core of what fuzzing is and precisely why it is so powerful. Many developers either don’t have enough time or simply forgot to write exhaustive test cases. Or, as is often the case, they are working on something so large with so many possible unique instances of interacting components that they cannot write tests that provide coverage of said instances. The irony -- or elegance -- here is to use a really basic technique to solve such a hard problem. You can start firing garbage input at your functions and seeing how they handle it. In fact, fuzzing as a field is the art of nuanced defect generation in inputs to computer programs. Like most fields, fuzzing had to start somewhere, and it started with the most basic case, which is random input and/or random alteration.

Corner/Edge Case Discovery

Let’s say you’ve written tests that do checks on data format in order to prevent incorrect data types or basic random garbage from crashing your code. By taking regular, expected to work inputs and randomly modifying portions you can create new test cases which provide

structured and expected input in most places but also have alterations in small regions. Bits could be swapped, added, and/or removed in this operation. In applying this operation, you've just generated new test cases on the fly. If you repeat this operation, you're generating multiple new test cases. You may also discover cases you have generated cases you hadn't considered in your testing, such as corner/edge cases. These are cases that only happen at extremes or outside expected operating parameters.

Purposely Malformed Input as a Means to an End

"They wanted to know how we were able to find so many flaws in Microsoft's products so readily […] [W]e replied […] we could input random garbage into the systems[.] […] We followed up by asking what sort of […] testing they performed. [...] When we asked Microsoft [...] why they did not [...] provide malicious input [...], the answer was, 'Why would a user want to do that?'"

- Mudge, Beautiful Security, p. 10

The above quote uses the question of why someone would supply purposely malformed input to a computer program practically like the punchline of a joke. However, most developers and many companies are focused on building software and building it quickly and with features. One of the things they often don't fully consider is security. Just as developers are building software, hackers are taking it apart. There are a whole bunch of reasons why people reverse engineer software, one of them being exploitation. While a seemingly militaristic and ominous term, the phrase -- in this context -- describes doing something that was not initially intended to be allowed by a piece of software. For instance, in an online banking system, I should not be able to add money I do not have to my account balance. In this case, doing so would be an instance of fraud and if I ever were to withdraw/transfer such funds that would be theft.

Software exploitation bridges the gap between “should not be able do” and “is able to do” on a computer. As long as there is a disparity between these two categorizations people may be likely to do what they are -- by intended definition -- not “allowed” to do for any number of motivations.

Automatic Discovery of Exploitable Vulnerabilities

In order to exploit software you need to find vulnerable portion of code which you are able to modify and eventually control through the supply of maliciously crafted input. If you are developing software you want testing techniques to prevent your software from being exploited. If you are analyzing software for exploitability you want a method for finding deviations from expected program function based on inputs you control.

There are, however, real-world boundaries that throw a wrench in conducting exhaustive, rigorous analysis for vulnerabilities. As an attacker, you are looking for a specific input that triggers a specific program state that you may be able to leverage in order to control execution. The question is what is a good approach for finding if these states even exist? Consider the following question: given a computer program **P**, where **P** reads an input **I**, is there an input **I** that causes **P** to crash? Now consider the same scenario with the following inquiry, is there an input **I** that results in state **S**? State **S** in this instance is a specific result state that is desired by an attacker. For instance this could be the spawning of an administrative shell by program **P**. You can try to answer the first question by reasoning that there are two consequential states for unique inputs **I**. The first is that **I** causes **P** to crash, the second is that **I** does not cause **P** to crash. These states are mutually exclusive. Thus, in the set **I**, there is a subset of elements which can be defined as those inputs that cause a crash. None of the elements of this subset are in the subset of elements in **I** defined as those elements that do not cause a crash [4].

So, it stands to reason, that you can test an input against that definition and that's it. Unfortunately, this is impossible. Consider an input **I** which results in **P** running forever. Now consider answering whether or not the input will crash the program. If you don't know a priori that the input will result in a termination state you never will because the program could terminate in the future or not. You're left running it forever and still unsure if it will terminate in the next second or never.

The space of possible inputs is also massive, which means that you cannot realistically even hope to try something that even approaches exhaustive search. So this is where fuzzing comes in to provide utility. Fuzzing is a crash discovery heuristic. Fuzzing becomes interesting when it is used to alter known working input because it is a vulnerability discovery technique that orients itself with respect to known non-vulnerable cases. The idea is that if you alter a case known to be working, you have a higher likelihood of triggering a crash than you do iteratively sending completely random input.

DeMott's Clusterfuzz

DeMott works primarily in Fuzzing and has written several very unique fuzzers I find very interesting. I was interested to see how he structured his approach to making fuzzing scalable. He started by selecting only one fuzzer, an open source fuzzer known as Peach fuzzer. Peach is a very reasonable choice and is one of the more popular fuzzers. Peach is configured with schema driven files that allow for highly detailed, very granular control over the entire fuzzing process. If no file was provided to Peach an empty one would be created that defaulted the run to mutation based fuzzing. DeMott used a VMware ESX cluster for virtualization on which he spent 30,000$.

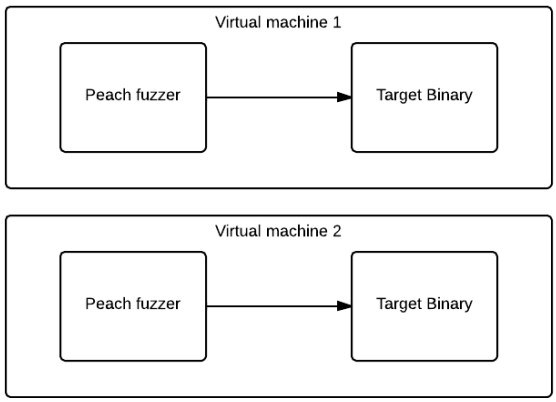
DeMott broke Clusterfuzz operations into three stages: configure, run, collect and report. In the configure stage, the user has to create a virtual machine image and install the target binary into the virtual machine. Next, the configuration file for Peach fuzzer can be uploaded or generated. In the run stage, runs are started and virtual machines are copied and run on the cluster. During the Collect and Report stage DeMott, reports crash runs from the Clusterfuzz GUI.

Exploring Alternatives in the Clusterfuzz Design

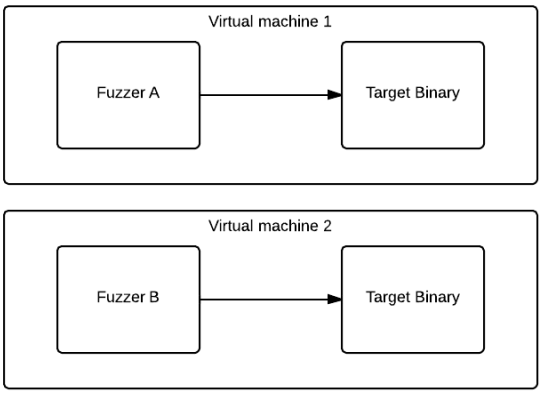
My interest was basically in replicating DeMott's work with a few -- key -- distinctions. The first thing I wanted was to test multiple fuzzers (not just PEACH) against a target binary. The second thing I wanted was to run this in the cloud, as I do not have a cluster. The third thing I wanted was an API that would allow me to create runs and get back crashes.

Benefits and Drawbacks of Multiple Fuzzers

By limiting the fuzzer involved, DeMott allowed his runs to have uniform configuration and reduced complexity in executing fuzzing runs. I wanted to design a system that could operate with multiple fuzzers on the same target binary. In the DeMott cluster fuzzing, Peach was used in every vm against the target. I wanted the user to be able to specify possibly more than one fuzzer and have fuzzing rounds take place at the same time on different virtual machines.



Peach fuzzer as only fuzzer in DeMott Clusterfuzz.



Multiple fuzzers in new fuzzing system.

Having the ability to automatically implement multiple fuzzers against the same target binary can provide a number of benefits. First, there are different fuzzing tools that try to generate input in different ways. Secondly, there is no apparent standard approach for deploying fuzzers that I am aware of and no automated way to quickly test one fuzzer against another on a certain program. For instance, say I want to compare the results of mutation, generation, and evolutionary fuzzing against the same binary or across a set of binaries and gather statistics about the runs. There is currently no way to do this other than by hand.

However, building a system that offers more than one fuzzer increases complexity. In the case of Peach fuzzer and a binary being fuzzed on an ESX cluster, you set it up once and then copy the virtual machine a given number of times. In this case, for a given target binary, there are fuzzers which can and fuzzers which cannot be used. This requires logic for identifying compatibility with a target binary as well as setting up each specific fuzzer to be used against the target binary.

This type of fuzzer matching can be very fine-grained and overlapping. Consider the scenario of an HTTP server and an IRC server and wanting to fuzz both. If you have a collection of available fuzzers, how do you match relevant fuzzers to both? One answer is to just let the user query a list of all fuzzers and select those relevant. Even then, however, there should be some control over querying a subset of fuzzers by a certain attribute.

In order to build an attribute list that is both extensive and easily modified, the concept of a tag cloud can be applied to each available fuzzer. That is to say, one approach is to let users of such a system define the relationships between fuzzers in whatever relationships they can specify. Having tags for fuzzers allows user to create tag based queries that result in fuzzer selection.

Tags in an Automated Exploit Pipeline

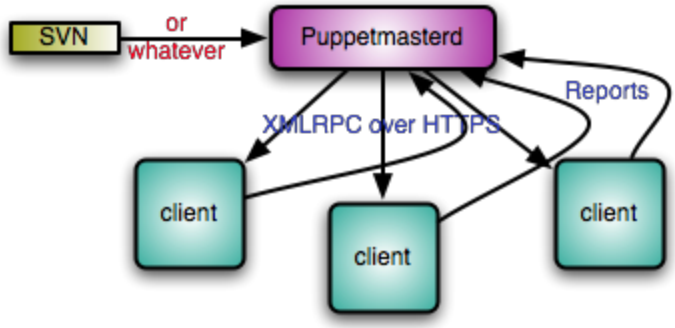
One of the useful features of tags in an automated exploit pipeline for fuzzer selection is the ability to automatically extract tags from binaries. For instance, there could be preprocessing software that extracts function calls from a binary and infers the types of inputs the binary can receive. This is a high level approach to figuring how to select fuzzers and apply them to certain programs. It is not implemented in this project.

Cloud Fuzzing Architecture

The initial proposed architecture for my Division 3 involved fuzzing on Amazon Web Services. Amazon provides cloud instances of remote virtual machines for running software in their own virtual machines. Since I did not have my own cluster, I planned on writing a system that used Amazon for hosting of virtual machines. Essential to my Division 3 is my ability to remotely install, configure, and run software. This applies to fuzzers as well as installing and

monitoring the target binary for crashes. In my initial architecture design I proposed using Puppet for machine configuration.

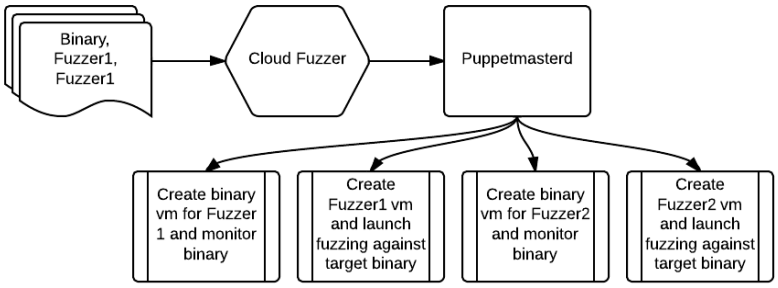
How Puppet Works



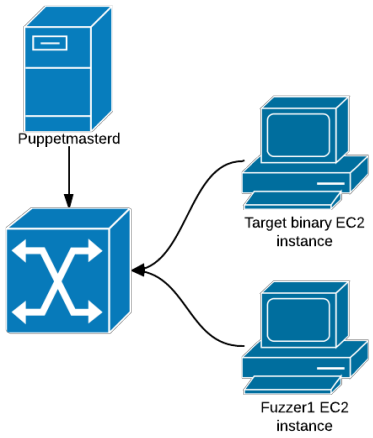
Puppet is a software suite that provides a domain specific language in addition to a client/daemon architecture for remote provisioning of machines in a programmatic matter. The Puppet daemon is known as the Puppetmaster. Clients are physical or virtual machines which have the Puppet client software installed. The domain specific language allows for the specification of system administration tasks on Puppet clients as well as system reporting.

Installing Fuzzers with Puppet in AWS

Operating on AWS, the Puppetmaster will be a virtual machine as will the clients. For every fuzzer in my proposed cloud fuzzer system, there would be a Puppet script which specified the system administration steps which constituted the installation and launch of a specific fuzzer against the target binary.



A user would submit a binary to be fuzzed along with a list of fuzzers to use to a CloudFuzzer application. The CloudFuzzer application would find records for each fuzzer with corresponding Puppet scripts. The CloudFuzzer would then communicate with the Puppetmaster daemon would then create new Amazon EC2 instances for fuzzing rounds as shown. Each created EC2 instance would have the Puppet client installed. EC2 instances would then be configured in order to setup necessary environments for fuzzing operations. Puppetmasterd would then launch the fuzzing operation.



Pairs of EC2 instances would then be connected using a virtual interface allowing the Fuzzer EC2 instance to reach the Target binary EC2 instance over the network using a local IP. Both machines would still be reachable by the Puppetmaster.

Provisioning a Fuzzer in a Cloud Fuzzing System - Examples

While prototyping this code I worked with manual fuzzing rounds in the beginning. The first fuzzer I chose to write an automated installation routine for was Dave Aitel's SPIKE fuzzer[5]. In this case, the assumption is that SPIKE has been selected by the user and must be provisioned. In this case the Kali GNU/Linux distribution (formerly known as BackTrack) comes

with SPIKE fuzzer installed and is available as an EC2 image as an Amazon Machine Image (AMI) [6]. Puppet has the capability of creating new EC2 instances from custom AMIs using the puppet node_aws create command [7]. By issuing the node_aws create command specifying the Kali AMI, an EC2 instance with SPIKE fuzzer installed has been created.

Provisioning a Target Binary EC2 instance in a Cloud Fuzzing System

This is where one of the biggest problems in running a fuzzing operation in the cloud occurs. When DeMott installed his target binary into a virtual machine on his ESXi cluster, I imagine that he replicated runs by creating a snapshot of the virtual machine directly after the target binary had been installed. Following this, he most likely replicated the snapshot into unique virtual machines to parallelize fuzzing runs. Because DeMott was running his own cluster, he could replicate a virtual machine running a proprietary Operating System as many times as he wished (bounded only by space and computing power). However, doing so on AWS infrastructure came at a financial cost until recently. Consult the architectural designs appendices for more info.

More on Fuzzing

When Fuzzing Should Be Performed

Fuzzing can be used as a Quality Assurance technique at any stage of the development and deployment lifecycle(s). However, it is worth nothing that fuzzing is a potentially disruptive form of software quality assurance. Due to this factor, fuzzing should be used carefully in production environments. However, that is not say that fuzzing is not regularly used in production environments. One common technique of attackers is to probe black-box, production deployments by experimenting with inputs and tracking the outputs. This, in and of itself, is a

primitive form of manual fuzzing by hand. In the case of real-life threat assessment in a penetration test, fuzzing can be used to show the potential consequences of having remotely "exploitable" bugs (even if the bug is limited to a denial of service or undesired taint/corruption) in production environments.

What is to be Fuzzed

While fuzzing has aspects which seem to relegate the craft to that of a black art, there are still rules. One of the most basic constraints is what is to be fuzzed. In order to perform fuzzing, there must be a consistent interface to a piece of software. This interface could, for example, be a file that is read by the program, an environmental variable, a network packet, or an audio stream. Once this input stream is known and specified, fuzzing takes place by either intercepting and modifying the input and the input submission to the interface with a modification procedure or an initial generation of purposely modified input.

From the standpoint of a malicious attacker, these inputs are known as the attack surface. However, the attack surface is not limited to inputs. Data extraction is a potentially consequence of fuzzing, but extraction points from an environment cannot be fuzzed if they are not dependent upon user created – or inadverdently user created – input.

Where to Fuzz

Fuzzing can be applied wherever computer programs that parse local or remote input run. One domain in which fuzzing has proved a very popular approach is web application vulnerability discovery. In "Fuzzing for Software Testing and Quality Assurance," the authors state, "[a]lmost 50% of all publicly reported vulnerabilities are related to various packaged or tailored web applications. Almost all of those vulnerabilities have been discovered using various forms of fuzzing." Files transferred over networks are also popular for fuzzing. Even network

protocols can be fuzzed. Wireless protocols (e.g. Bluetooth) have also proven a popular realm for fuzzing in recent years.

Why there are bugs to be Fuzzed

Bugs are due to the human element, a given environment, or a combination thereof. Rushed time to market for a piece of software can oftentimes cause reductions in Quality Assurance testing. Another common reason that exploitable bugs exist in software is that many developers don't write code with the consideration of defense against a malicious attacker. Software can become very complex very quickly, and as complexity increases the likelihood of a bug increases as well. Takanen, et. al provide the example of a thousand lines of code between two to ten critical defects. Even with these conservative estimates, products with millions of lines of code can contain thousands of flaws that have not been uncovered.

In the context of an attack surface, a malicious attacker doesn't necessarily care about the popularity of the vector they are exploiting. However, testing is oftentimes proportional to feature popularity. For instance, Microsoft has been subject to numerous exploitable vulnerabilities in the product Microsoft Word through the attack vector of maliciously crafted Rich Text Format (RTF) files. These are legacy files which are seldom used by consumers of Microsoft Word as a default file format (DOCX is the default extension). However, due to the fact that they are a potentially fruitful attack vector to a malicious attacker they have been – most likely – fuzzed and exploitable vulnerabilities have been identified. When a software development or quality assurance team is faced with delivering an update or new feature set, they can possibly not test thoroughly or write tests that provide good code/case coverage. Other possible reasons for bugs include feature creep and design by committee.

Understanding Fuzzing by Example

This entire example is based on the YouTube video series "From Fuzzing to Metasploit" by Andew Whitaker. It was independently replicated. Images are my own.

In order to illustrate the utility of fuzzing, we can work through an example scenario using an application that is known to be vulnerable. In this example, we will be using the vulnserver application. Stephen Bradshaw, the author of vulnserver, describes the tool as "a Windows based threaded TCP server application that is designed to be exploited."[?] This tool is useful for illustrating vulnerability discovery techniques because it is known to be vulnerable. Using vulnserver, we can walk through the process of setting up a vulnerable application, finding a crash through fuzzing and replicating the crash.

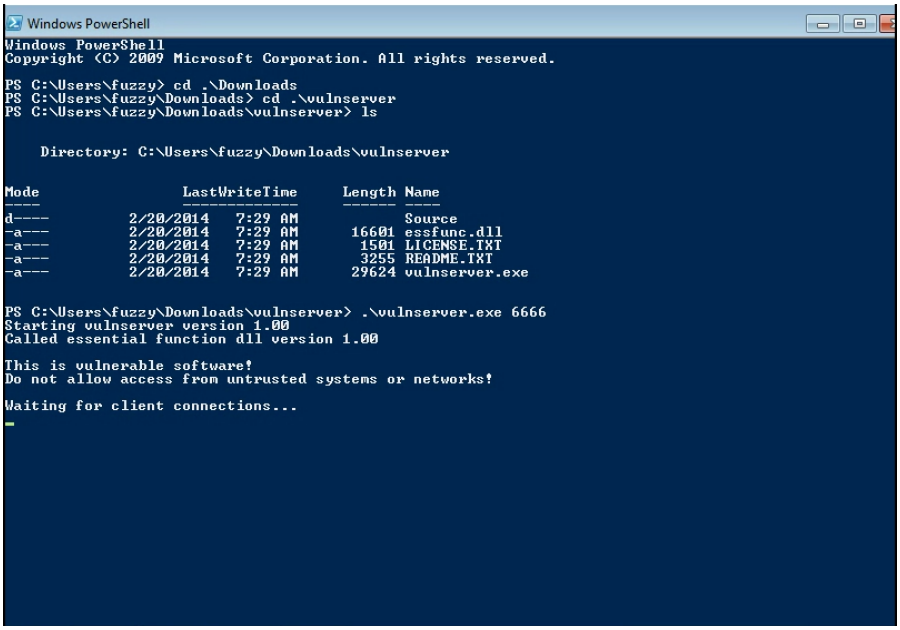
Setting up Vulnserver and Required Environment

In order to demonstrate fuzzing vulnserver we will require a network with two machines. The first machine will be a Microsoft Windows machine that runs vulnserver exposed over the network. The second machine will be that of the attacker and will be on the same network. The attacker machine will run the fuzzer code. When constructing this walkthrough, Oracle Virtualbox was used to create two virtual machines with bridged network connections for connectivity between one another. Windows 7 64 bit was used as the target base, and Kali Linux was used as an attacker base[1]. The Windows 7 vm requires vulnserver and ollydbg to be installed.

Download and Run Vulnserver

[1] Kali Linux is a GNU/Linux distribution based on Debian GNU/Linux and geared towards penetra- tion testing and vulnerability research.

Once a network with both a target and attacker machine is in place, vulnserver can be downloaded onto the target machine[2]. After having downloaded vulnserver, it can be unzipped and executed. If you have a firewall running, make sure to allow vulnserver to receive remote connections.



Test Vulnserver Connections from Kali

In this setup, vulnserver is running on port 6666. From the Kali Linux machine, a connection can be tested using the netcat command like so (assuming the IP address of the machine running vulnserver is 192.168.56.102):

Vulnserver should log the connection like so:

[2] There is nothing that prevents this from all taking place on one machine. However, Kali Linux provides an environment that reduces preparation in performing these types of tasks.

Typing HELP in the vulnserver session list available commands:

Setting up SPIKE for vulnserver command fuzzing

When thinking from the point of view of an exploit developer, these commands are all potentially exploitable buffers. A fuzzing application can now be used to fuzz the buffers in an attempt to get the vulnserver to crash. The fuzzer used in this example is the SPIKE fuzzer, written by Dave Aitel[?]. One way of interacting with SPIKE fuzzer is by using its built in strings for fuzzing. The following script is a SPIKE configuration file which fuzzes the parameter to the STATS command in vulnserver. It can be saved with the file extension .spk and provided to the SPIKE fuzzer as a command line argument.

```
s_readline ();
s_string(''STATS '');
s_string_variable(''COMMAND'');
```

SPIKE can now be called to attempt to fuzz the STATS command parameter in the vulnserver app using the above spk file.

```
root@kali:~# generic_send_tcp 192.168.56.102 6666 /root/vulnserver.spk 0 0
```

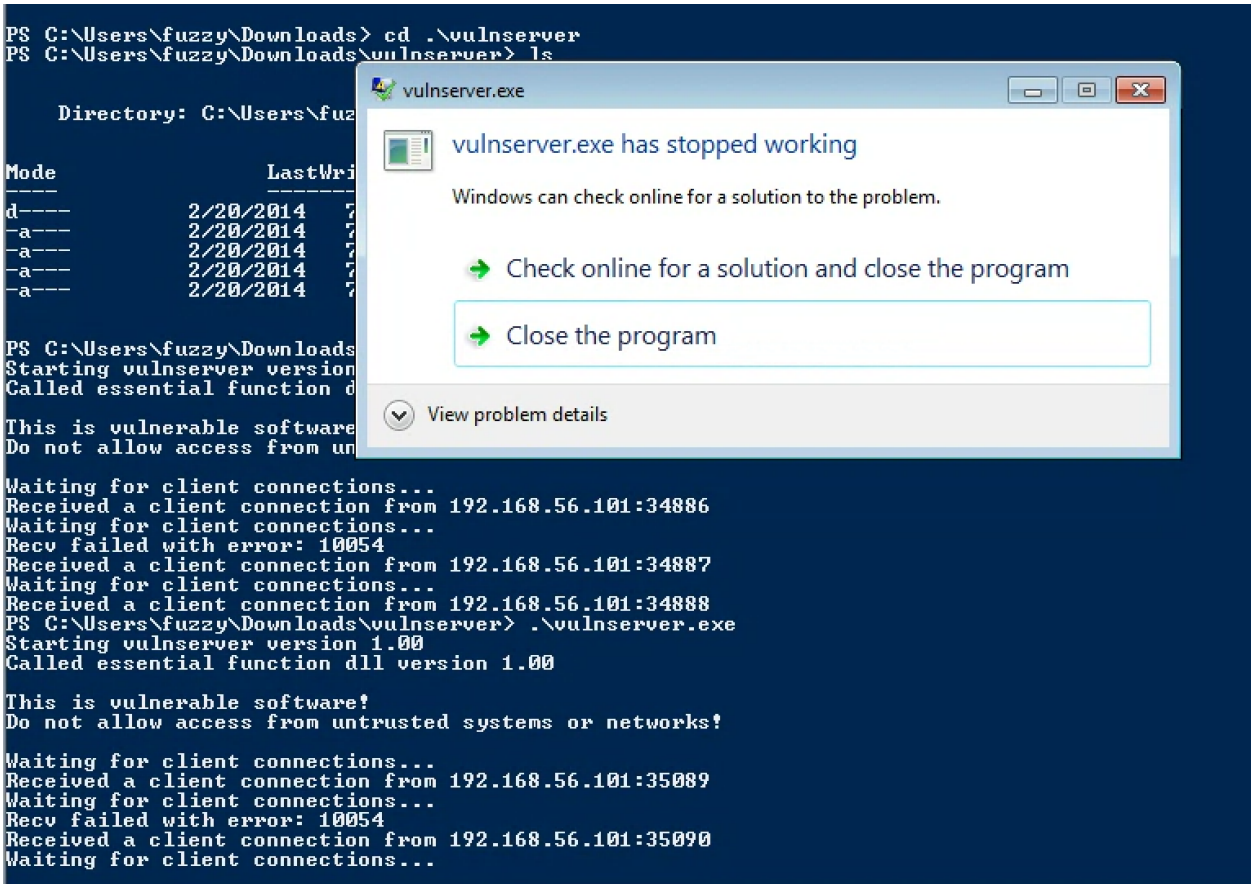
Upon issuing this command, the spike fuzzer runs with the following output:

The last line in the output is the line presented by vulnserver while it waits for user input. This means that SPIKE fuzzer was unsuccessful producing a crash by fuzzing the STATS command parameter. Another way to verify whether or not the application has crashed is to check the Windows virtual machine in which vulnserver is running.

At this stage an attacker could choose to fuzz a different parameter. In this case let's choose to fuzz the parameter of the TRUN command in vulnserver. We can modify our STATS fuzzer like so:

```
s_readline ();
s_string(``TRUN ``); s_string_variable(``COMMAND'');
```

When we run the SPIKE fuzzer with all the same parameters with a configuration file modified to fuzz the parameters of the TRUN command, we see that SPIKE triggers a crash in vulnserver.



From having observed the SPIKE fuzzer we see that the fuzzing operation appears to consist of trying multiple inputs. There is now a question of which input led to a crash.

Identifying fuzzed requests from SPIKE that triggered a crash

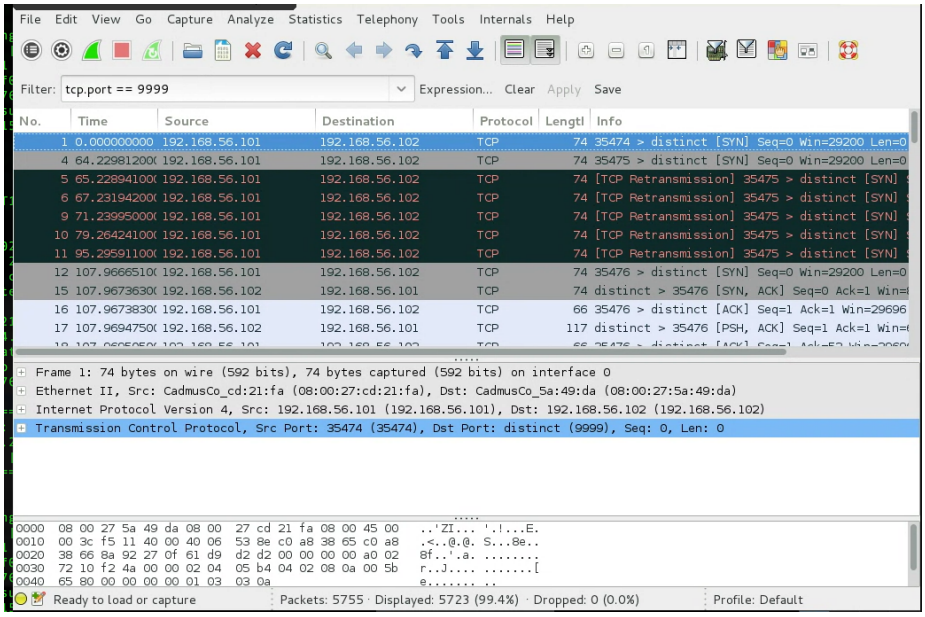
The SPIKE fuzzer performs the task of parsing the fuzzer configuration file and modifying requests over and over again. It does not know whether or not a certain request triggered a crash. However, during the course of the fuzzing run, we observed that the vulnserver instance crashed during the fuzzing of the TRUN parameter. Since this fuzzing task is taking place over a

network, we can observe the network traffic emanating from the Kali Linux virtual machine to isolate the input which resulted in vulnserver not returning the banner which indicates it successfully parsed input. Failure to present a banner, in this instance, is a good potential indicator of a crash.

Using Wireshark to Inspect Network Traffic

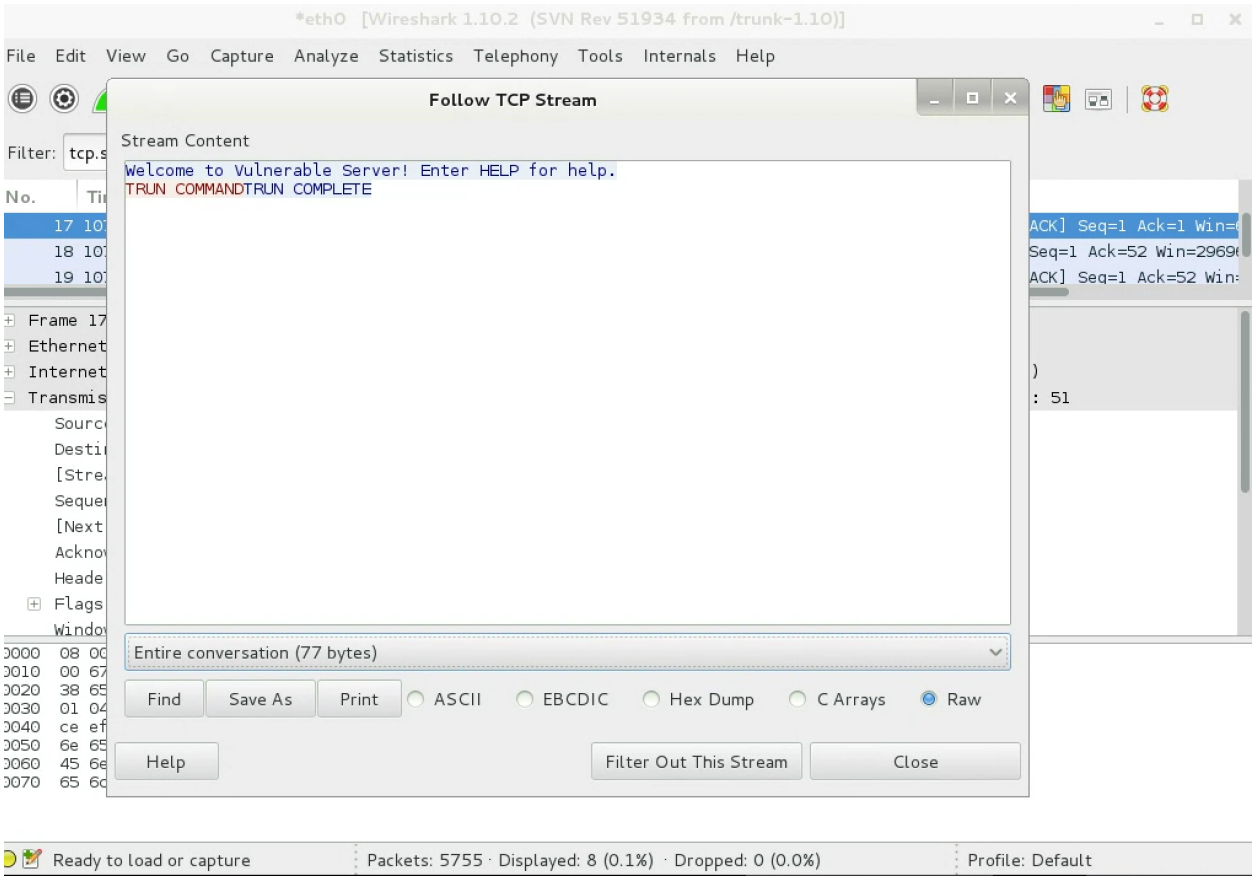
It is possible to inspect network traffic to identify the crash producing input with a packet sniffer on either host. We will be using the wireshark network inspection tool on the Kali Linux machine for two reasons. The first reason is that the tool ships with Kali Linux. The second reason is that it is more realistic for the analysis of a crash producing input to take place from the machine of an attacker, since prior to breaking into a remote instance of an application it can be difficult to setup a network packet sniffer on that machine.

The wireshark tool should be run on the interface which is sending traffic to the remote vulnserver instance. In our case – and in most cases – this interface is eth0. Now, in order to capture the crash producing input, we have to run the fuzzer again with the packet capture utility in place. We can use a wireshark filter to limit the traffic to packets being sent to the vulnserver port using a tcp port filter.

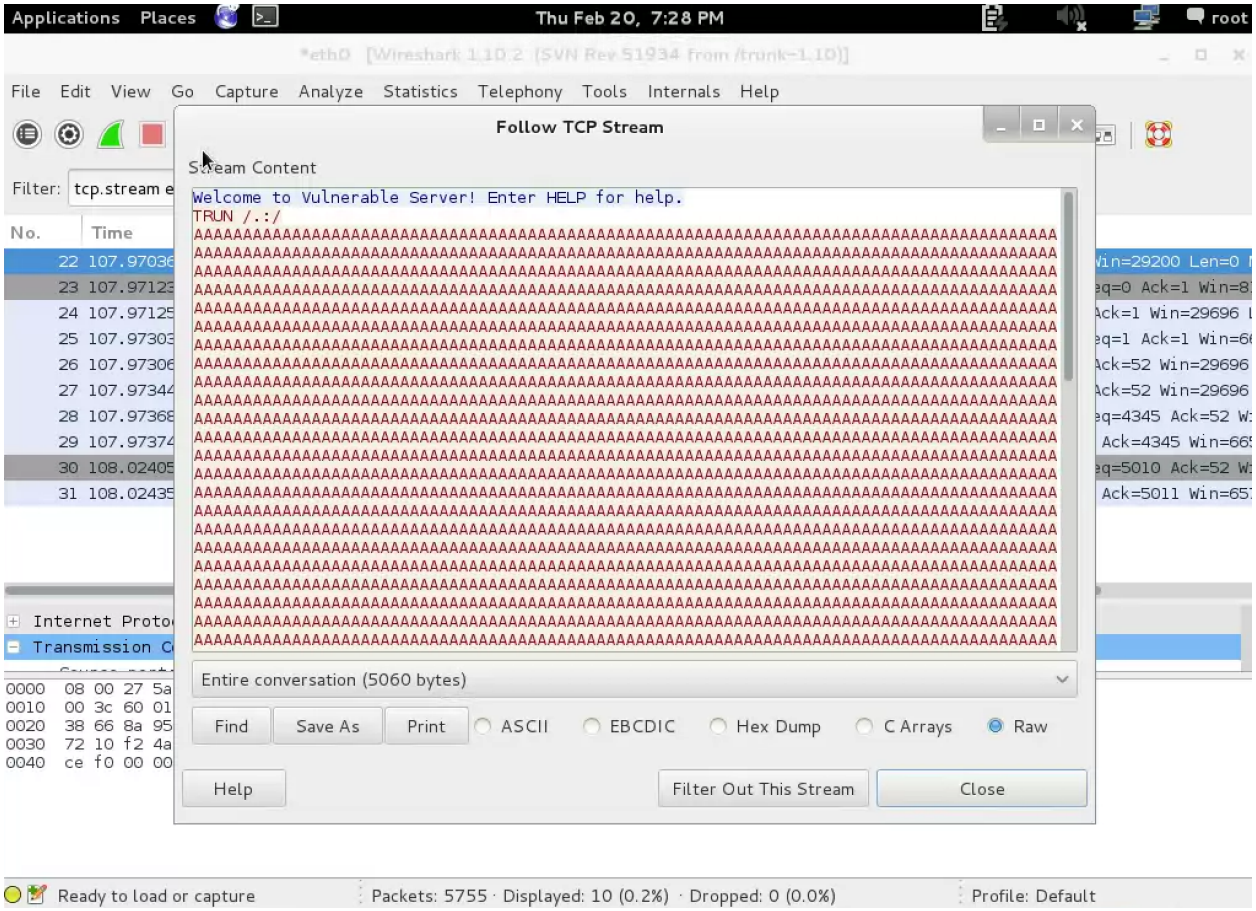


Now we can examine the packets from first to last until we find the packet where input is sent to vulnserver and no banner is received. We can select TCP packets and reassemble the TCP stream to build examine client to server communication. In the following graphic, we see SPIKE requesting the TRUN command and the server responding. In this TCP stream, there is no crash.

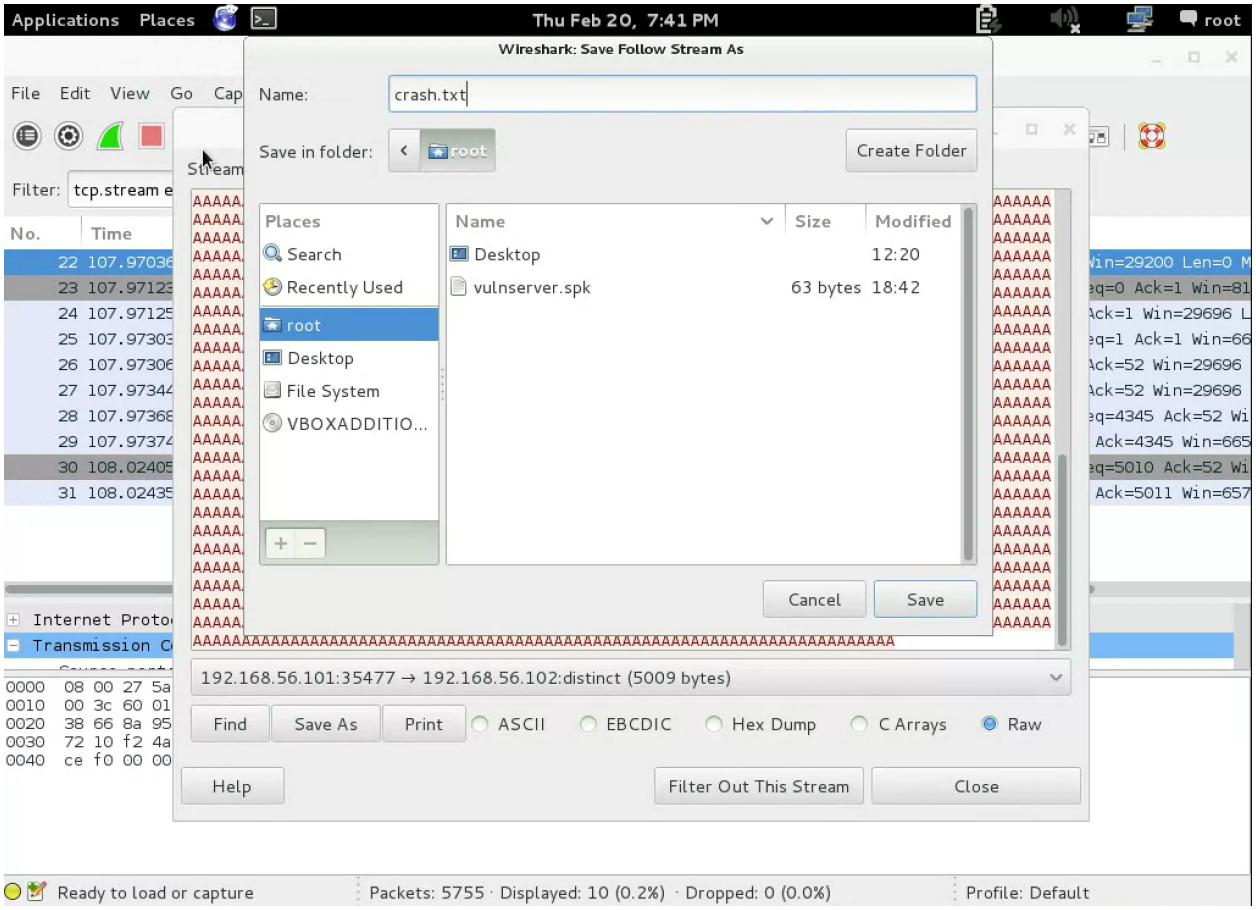
We can work through reassembled TCP streams until we find a session where the TRUN command does not respond. In the following TCP stream reassembly, we see that the SPIKE fuzzer has constructured a very large input parameter to the TRUN command.



Following this TCP stream reassembly to the end of the input we see that the TRUN command was not returned.



This TCP stream contains the crash producing fuzzed parameter to the TRUN command constructed by SPIKE. We can now save the outbound portion of the TCP stream reassembly to a text file to perform subsequent analysis on the crash producing input.



End of section taken from Youtube series

Types of Fuzzing

There are two main types of fuzzing: generation and mutation based. Miller, et. al, also term these techniques as intelligent and non-intelligent, respectively. Mutation based fuzzers can generally be characterized as unaware of the underlying structure of data being fuzzed. For instance, if the fuzzing target is an image of a certain format, a gen- eration based fuzzer will ignore format structure in the output generation. Generation based fuzzers tend to be targeted at a specific format or protocol.

Mutation Based Fuzzing

Mutation fuzzers are also called “dumb” fuzzers. Consider the case of file format fuzzing. One good example of a mutation based fuzzer is the zzuf tool[3].

```
zzuf is a transparent application input fuzzer. Its purpose is to find bugs in
applications by corrupting their user-contributed data (which more than often
comes from untrusted sources on the Internet). It works by intercepting file
and network operations and changing random bits in the program’s input. zzuf’s
behaviour is deterministic, making it easier to reproduce bugs. Its main areas
of use are:

      - quality assurance: use zzuf to test existing software, or integrate it
into your own software’s testsuite

      -security: very often, segmentation faults or memory corruption issues
mean a potential security hole, zzuf helps exposing some of them

      -code coverage analysis: use zzuf to maximise code coverage

zzuf’s primary target is media players, image viewers and web browsers,
because the data they process is inherently insecure, but it was also
successfully used to find bugs in system utilities such as objdump.

zzuf is not rocket science: the idea of fuzzing input data is barely new, but
zzuf’s main purpose is to make things easier and automated.
```
- [8]

Zzuf is quite literally just a tool that randomly flips bits in data, yet it has successfully produced crashes in software such as VLC media player, mplayer, and GStreamer. To demonstrate the power of zzuf, we can use it to try and crash some software.

Finding a Double Free Crash for a Linux PDF reader in under 15 seconds

Zathura is a keyboard driven pdf viewer for GNU/Linux systems. We will be using zzuf to fuzz an input pdf file with random modifications by specifying a seed value. We specify the number of simultaneous jobs with the -j flag, the -v flag for verbose output, the -q to hide the output of the fuzzed application, the -c flag to specify the commandline parameter to the binary is to be fuzzed, the -r flag to specify the minimum to maximum proportion of the file to be fuzzed,

[3] This is the word fuzz backwards.

and the -s flag to specify the random seed to use for a deterministic fuzzing run. This run was created and run by the author with a PDF seed file he did not create.

```
$ zzuf -j2 -vqc -r0.0001:0.01 -s0: zathura Evolutionary-Fuzzing.pdf
zzuf[s=0,r=0.0001:0.01]: zzuf[s=1,r=0.0001:0.01]: zzuf[s=1,r=0.0001:0.01]:
zzuf[s=2,r=0.0001:0.01]: zzuf[s=0,r=0.0001:0.01]: zzuf[s=3,r=0.0001:0.01]:
[snip]
launched launched exit 1 launched exit 1 launched
‘zathura ’ ‘zathura ’
‘zathura ’ ‘zathura ’
zzuf[s=66,r=0.0001:0.01]:
*** Error in ‘zathura’: double free or corruption (fasttop):
0x0000000001bfba60 ***
======= Backtrace: =========
/usr/lib/libc.so.6(+0x731ff)[0x7f2ab2a541ff]
/usr/lib/libc.so.6(+0x789ae)[0x7f2ab2a599ae]
/usr/lib/libc.so.6(+0x796b6)[0x7f2ab2a5a6b6]
/usr/lib/zathura/pdf.so(pdf_page_clear+0x65)[0x7f2aaacbede5]
zathura(zathura_page_free+0x40)[0x427670]
zathura(zathura_page_new+0x72)[0x427712]
zathura(zathura_document_open+0x19e)[0x42489e]
zathura(document_open+0x51)[0x420891] zathura[0x4212d2]
/usr/lib/libgdk -3.so.0(+0x1f678)[0x7f2ab4577678]
[snip]
```

Through only random input file modification by zzuf to Zathura, we have found an instance where a pointer appears to be accidentally freed twice. This may or may not lead to an exploitable vulnerability. However, the ability to quickly discover a crash that may lead to a potential exploit in such a short period of time is impressive.

malloc() SIGSEGV in a Linux CHM viewer in under 5 seconds

The next example is fuzzing a Linux reader for Microsoft Compiled HTML Help files (CHM). This format is popular for online books and is similar to pdf and epub. In this run we use the same command line arguments but specify a the target binary to be kchmviewer (the linux CHM reader). This run was created and run by the author with a CHM seed file he did not create.

```
$ zzuf -j2 -vqc -r0.0001:0.01 -s0: kchmviewer \\
~/Downloads/chainedExploitsAdvancedHacking.chm
zzuf[s=0,r=0.0001:0.01]: launched ‘kchmviewer ’
```

```
zzuf[s=1,r=0.0001:0.01]: launched 'kchmviewer '
*** Error in 'kchmviewer': malloc(): memory corruption (fast):
0x00000000011dfe60 ***
exit 255
launched 'kchmviewer ' signal 11 (SIGSEGV)
zzuf[s=0,r=0.0001:0.01]:
zzuf[s=2,r=0.0001:0.01]:
zzuf[s=1,r=0.0001:0.01]:
*** Error in 'kchmviewer': malloc(): memory corruption (fast):
0x0000000001e01ed0 *** zzuf[s=2,r=0.0001:0.01]: signal 11 (SIGSEGV)
zzuf: maximum crash count reached , exiting
```

Zzuf produces a crash in the kchmviewer application after only a few runs and denotes that the application crashes with a Segmentation Fault. Again, this may or may not lead to an exploitable vulnerability after vulnerability analysis.

Generation Based Fuzzing

Generation fuzzers are also called intelligent fuzzers. Generation based fuzzers rely on prior knowledge of a protocol or file format.

One fuzzer example is Peach fuzzer. Peach possesses both mutation and generation based fuzzing capabilities. The fuzzer can be scripted using XML based configuration files, which are known as “PeachPit” files. The XML files contain the required information about what is to be fuzzed (e.g. data structure, type information, relationship of data).

Performing intelligent fuzzing requires much more preparation as the fuzzer needs to have some knowledge about the data format and what is to be fuzzed. The PeachPit file addresses this requirement in its five-part structure:

- DataModel - contains the structural definition of the data.
- StateModel - manages the data flow during a fuzz process.
- Agents - monitor application behavior during fuzz process.
- Test Block - builds test cases from DataModel, StateModel, and Agent.

- Run Block - defines which tests will be run.

The ZIP fuzzing example and explanation of PeachPit file can be further researched at http://www.flinkd.org/2011/07/fuzzing-with-peach-part-1/, which is also from where the bullet point list was extracted and paraphrased. However, the document refers to an older version of Peach fuzzer with a different syntax for PeachPit files.

Peach provides templates for building a fuzzer for a specific protocol. Included with the Peach source code is a sample PeachPit file called template.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach" \
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" \
xsi:schemaLocation="http://peachfuzzer.com/2012/Peach /peach/peach.xsd">
  <!-- TODO: Create data model -->
  <DataModel name="TheDataModel">
    <!--<Blob/>-->
  </DataModel>
  <!-- TODO: Create state model -->
  <StateModel name="TheState" initialState="Initial">
    <State name="Initial">
      <Action type="output">
        <DataModel ref="TheDataModel"/>
        <!--<Data fileName="samples_png" />-->
      </Action>
      <!--<Action type="close"/>-->
      <!--<Action type="call" method="ScoobySnacks"
publisher="Peach.Agent"/>-->
    </State>
  </StateModel>
  <!-- TODO: Configure Agent -->
  <Agent name="TheAgent" location="http://127.0.0.1:9000">
    <!--<Monitor class="WindowsDebugger">
        <Param name="CommandLine" value="mspaint.exe fuzzed.png" />
        <Param name="WinDbgPath" value="C:\Program Files (x86)\Debugging Tools
for Windows (x86)" />
        <Param name="StartOnCall" value="ScoobySnacks"/>
        </Monitor>
        <Monitor class="PageHeap">
        <Param name="Executable" value="mspaint.exe"/>
        <Param name="WinDbgPath" value="C:\Program Files (x86)\Debugging Tools
for Windows (x86)" />
        </Monitor>-->
  </Agent>
  <Test name="Default">
    <Agent ref="TheAgent"/>
    <StateModel ref="TheState"/>
    <!-- TODO: Configure a publisher -->
```

```
<!--<Publisher class="File">
    <Param name="FileName" value="fuzzed.png"/>
    </Publisher>-->
```

A PeachPit modified for the ZIP file format which is written to fuzz portions of data can be found in Appendix C. The sheer specificity of this file and its conformance to the ZIP specification should illustrate the difference between mutation and generation based fuzzing.

Ultimately, the difference between generation and mutation based fuzzing techniques is interface knowledge. A mutation based fuzzer has virtually no interface knowledge while a generation based fuzzer has greater interface knowledge. For fuzzing tools, there is a gradient between these two definitions where many tools reside.

Mutation based fuzzers can, in many instances, be characterized as having low code coverage. Intuitively, this makes sense, as software that implements basic checks will most likely discard many inputs from a mutation based fuzzer. Consider the dependence of a decompression in a ZIP file extractor based on calculating the CRC-32 checksum as noted in the ZIP specification. If a checksum operation fails, the code will (possibly) terminate with a corrupted archive error. Now consider the same scenario with a gener- ation based fuzzer. Because the fuzzing approach was built around the specification, the region of the file which corresponds to the CRC32 of another region of the file has been preserved in the fuzzing routine. When a file is fuzzed, if a region that is checksummed is touched, the subsequent checksum will be modified. Taking this approach allows for greater code coverage by increasing the likelihood of basic error detection checks in software.

EC2fuzz

EC2fuzz is the code I wrote for my Division 3. It automatically configures an Amazon EC2 Amazon Machine Image (AMI) for being suitable for fuzzing, snapshots it, and uses that snapshot to fuzz binaries which can be loaded onto the instance remotely at runtime. It is capable of scaling fuzzing runs by replicating the machine an arbitrary number of times. EC2fuzz is a prototypical cloud fuzzer written in Bourne shell and runs on POSIX compliant systems (e.g. Mac OS X, GNU/Linux).

The goal is to create a vm as a fuzzing environment, supply it with the necessary parameters, and start fuzzing. It completes this goal. It allows for run scalability and parallelization of runs. It is relatively cheap because it is done on virtualized infrastructure hosted by a third party provider. There is primitive reporting built in through a structured virtual filesystem mountpoint of remote filesystems containing fuzzer run logs. It can provision Windows environments in an automatic manner requiring no user intervention running as a BASH script on a client computer. It sets up all AWS dependencies for performing vm creation, vm state snapshotting, and vm configuration. It performs all the operations mentioned to setup fuzzing environments and deploy fuzzing runs in an automatic fashion.

EC2fuzz sets up an Amazon AWS security group and allows a client machine access to that security group. It creates and runs a stock Windows Server vm from AWS. This vm is configured to a specific state suitable for fuzzing. This state is then snapshotted into a new vm. This new vm is used as a base fuzzing image. The base fuzzing image is launched into a vm and a fuzzer configuration files and seed files are installed. A fuzzing run is then launched on the remote machine. The remote machine is mounted locally and the results of the run are read locally. This process can be done in parallel with multiple vms all using the same configuration and launched simultaneously.

EC2fuzz relies on a configuration file (config.sh), that specifies configuration. In the configuration file you can specify the base AMI using the `BASE_AMI` variable. The `BASE_AMI` variable corresponds to an Amazon Marketplace AMI id which represents a virtual machine disk image to be launched in EC2. EC2fuzz has been tested and runs with Windows Server 2008 r2 64-bit AMI images. The configuration files allows for the configuration of Amazon EC2 Security Groups with the variables `KEYPAIR` and `SECURITY_GROUP`. `KEYPAIR` specifies the file which will store an OpenSSL private key which will be used to make authenticate to the Security Group in order to make changes. `SECURITY_GROUP` specifies the name of the EC2 Security Group to be used.

When a Windows AMI is loaded, it is possible to provide runtime configuration during the instance boot period by specifying the configuration as a parameter named '`userdata`.' The configuration file allows the specification of instance userdata to be read from a file. The `USER_DATA_FILE` variable specifies the path to the file containing the userdata configuration. In EC2fuzz, the userdata configuration automatically downloads a subsequent configuration sequence from a remote server over HTTP. The configuration file allows the specification of this remote location using the variable `USER_DATA_URL`.

Everytime an EC2 AMI boots, a randomized password is generated for the Administrative account on that instance. EC2fuzz will override the randomized password with a prespecified password by inserting the password and required modifications into `USER_DATA_FILE`. The password to be inserted into `USER_DATA_FILE` can be specified using the variable `AMI_PASSWORD`. EC2fuzz mounts remote instances as local filesystem mountpoints as Samba shares over an SSH tunnel to provide access to fuzzing run logs and upload fuzzing configuration and seed fuzzing files. The location of the mountpoint root node under which all

instances will be mounted can be specified in the configuration file using the BASE_MOUNTPOINT variable.

If a custom instance has already been configured and snapshotted into a custom AMI, the provisioning step can be skipped when running EC2fuzz using the CREATE_AMI variable in the configuration file. CREATE_AMI is a boolean and, when false, will instruct EC2fuzz to skip the custom AMI creation phase. EC2fuzz has the ability to fuzz binaries already installed in the Windows operating system as well as other binaries which can be uploaded by the user. If the fuzzing run requires an external binary, it must be copied over to the instance. The necessity of this operation can be specified in the configuration file using the variable BINARY_INTERNAL. BINARY_INTERNAL is a boolean, and when true it will skip the binary copy step as it is not necessary.

If you choose to skip the custom AMI creation step when running EC2fuzz, you must specify the AMI id you will be using during fuzzing operations. This can be specified in the configuration file using the variable ami_id. To specify the number of instances to launch in a fuzzing run, you must set the NUM_INSTANCES variable in the configuration file. NUM_INSTANCES stores the number of instances to be launched. AWS requires users to specify the region their instances are running in as a prerequisite to the AWS command line tools functioning correctly. The configuration file contains an export of the EC2_URL environmental variable in case the script is being run in a standalone environment where the AWS command line tools have not already been configured after installation. EC2_URL specifies a URL for the instance location. An example URL used in the northeastern United States is https://ec2.us-west-2.amazonaws.com.

When these variables have been set, the EC2 script can be launched. EC2fuzz, at the time of this writing, accepts the command line parameter '-e' which corresponds to a full

End-to-end (e2e) test of EC2fuzz. The EC2fuzz e2e starts by executing a main function which calls a configuration loader which reads in the configuration file and loads the variables. If the file is cannot be found in the same directory under the name '`config.sh`' EC2fuzz will throw an error and exit. EC2fuzz then performs a dependency check to make sure the environment contains the required dependencies for the script to execute. EC2fuzz checks the environment for the GO programming language, which is a prerequisite for the '`winrm`' command. The winrm command is used for remote command issuance to Windows machines the UNIX client used in EC2fuzz is written using GO. EC2fuzz checks for the 'aws' command, which is a client tool suite for interacting with AWS on UNIX systems. EC2fuzz checks for the existence of the curl command for remotely getting the /24 network block of the client machine in order for security group access configuration. EC2fuzz checks for the existence of the ssh command as it is required for constructing an SMB tunnel between the instance and the client for mounting the instance as a local mountpoint. EC2fuzz checks for the sshpass command as it uses password parameter expansion in ssh connections to the instance.

EC2fuzz then creates a quoted and dequoted version of the password. The quoted version of the password is used for regex driven insertion into the userdata file. EC2fuzz then calls the `create_custom_ami` function, which setups the required security group, network routes, and provisions and snapshots a customized fuzzing instance of the Windows Server 2008 r2 AMI. The first step in custom ami creation is the creation of a keypair to be used in security group configuration and management using the `create_keypair` function. The `create_keypair` function checks for the existence of a file called 'keypair' in the local directory. If it is not found, an aws remote call is issued creating a keypair and registering it with AWS. This call returns a public and private keypair which is stored to the file `keypair` in the local directory. The private key portion is then extracted from the keypair file and saved in a file

$KEYPAIR.pem, substituting the keypair configuration variable as the name of the keypair private key file. The private key permissions are then restricted to be accessible only by the user running EC2fuzz.

EC2fuzz then calls the `create_security_group` function, which creates a security group on AWS tailored to the fuzzing run. EC2fuzz checks if the security group defined in the configuration file exists. If it does not exist, EC2fuzz creates a security group with the security group name specified in the configuration. EC2fuzz then calls the `authorize-ports` function, which enables network access over certain ports for the security group as well as establishes the IP address to specify when adding these routes. The first function called is `get_ip_cidr`, which downloads the visible user IP from http://checkip.amazonaws.com. The IP address is then transformed into a /24 network block and saved to a file named 'cidr' in the local directory. EC2fuzz then calls the `enable_rdp` function, which adds a route for the Remote Desktop Protocol to the security group. EC2fuzz then calls the `enable_winrm` function, which adds a route for the Windows Remote Management protocol to the security group. EC2fuzz then calls the `enable_ssh` function, which adds a route for the Secure Shell protocol to the security group. This route is a tunnel for the Samba protocol, which is blocked by most ISPs for security purposes.

Following security group configuration, the custom AMI password is then written into the userdata file to be sent to the AMI image for custom configuration during boot. The url for callout during the boot process from the userdata file is then written into the userdata file. Once the userdata file has been properly configured, the `launch_wintemplate_instance` function is called to launch the instance and perform the required configuration. The first step in the configuration is to locate and terminate existing template instances to reduce compute time and prevent a collision from having multiple instances with the same labels. EC2fuzz then launches

an EC2 instance based on the base AMI id described in the configuration file, specifies the instance type as a micro instance, provides the name of the public/private keypair, specifies the security group, and expands the output of the userdata file to be read by the instance upon boot. The returned instance id is then extracted and stored locally. EC2fuzz then applies the tag '`fuzzingtemplate`' to the EC2 instance so it can be referenced by this tag later in the script.

The next function called is `wait_for_instance` setup. This function periodically polls the system log of the instance and waits for an indication that the userdata file is being read by the loader. Once the userdata file has been parsed by the instance, the `test_winrm` command is called. This command extracts the publicly accessible IP of the instance and calls the `poll_config_progress` function. When the userdata file is parsed, it downloads a configuration file and logs the status of running the local configuration sequence as specified in this file. The winrm command is used to issue a remote command to the instance and return the output of the log file locally. The log file is then tested for the existence of the string 'Restarting', which indicates the configuration has been completed. This polling periodically repeats until the string is found.

Once the `create_custom_ami` function has completed, the `bake_instance` function is called. This function snapshots the current instance state into a custom AMI now that it has proper configuration to perform a fuzzing run. The first step of `bake_instance` is a call to `deregister_previous_ami`. This function will delete all AMI instances with the tag '`fuzzing-node-base`,' which is used to denote AMI images created by the `bake_instance` routine. The aws command '`create-image`' is executed with parameters specifying that the instance to be cloned is the instance created during `create_custom_ami`. The resulting AMI id is then extracted and stored.

After a custom AMI has been configured, the `launch_fuzzing_runs` function is called. This function reads the number of fuzzing run instances to be created, creates an array to store information about each fuzzing array, and calls the `launch_fuzzing_run` function for every fuzzing run to be launched. The `launch_fuzzing_run` function checks if an internal or external binary is to be fuzzed and then proceeds. The next step is a call to the function `launch_fuzzing_instance`, which will launch a customized fuzzing instance created by `bake_instance`. This command will check that the custom AMI is ready for through a call to the `check_ami_ready` function. This function queries the EC2 image description for the AMI and extracts the status field. If the status field is '`pending`,' the function will periodically poll until the status field has changed from '`pending`.' This command is used due to the snapshotting process taking some time to complete.

Once the AMI is ready for use, the fuzzing instance is launched specifying the AMI id, the instance type, the security group, and the keypair. The id of the instance is then extracted and stored. The IP address of the fuzzing instance is then extracted using the `get_fuzzing_instance_ip` function. This function polls the line corresponding to '`NICASSOCIATION`' in the fuzzing instance description. This line contains IP addresses attached to the virtual network interface of the instance. This command periodically polls the instance description until an IP address has been attached to the virtual network interface. The next function called is `check_fuzzing_instance_ready`, which periodically polls the console output of the instance until the string '`Windows is Ready`' is found in the log.

Once the fuzzing instance AMI has been launched, is accessible, and the boot process has completed, the instance is then mounted on the local filesystem of the client machine executing EC2fuzz using the `mount_instance` function. This functions initiates an SSH connection using the `create_ssh_tunnel` function. The first step in `create_ssh_tunnel`

is the generation of a random port for the tunnel port on the local machine. This is done in the `generate_random_ssh_port` function. This function generates a random number between 1024 and 49151, which is the range of ports that can be accessed on a local UNIX system without requiring root permissions. EC2fuzz then checks if the port is already in use by checking if there is a process already bound to that port. If there is a process running on that port, another random port is generated until an unused port is found.

An ssh tunnel to the remote instance is then initiated and the password is piped into the ssh process by wrapping the tunnel initiation with the `sshpass` command (external binary). The `sshpass` command can handle the ssh processes query for a password while it runs. The '`KnownHosts`' parameter of the ssh connection is specified to `/dev/null` to prevent a fingerprint for the specific IP address from being pushed to `~/.ssh/known_hosts`. This prevents errors due to mismatching host public keys between EC2fuzz runs. The '`StrictHostKeyChecking`' parameter is set to '`no`' to prevent a previously unseen public key fingerprint from preventing successful tunnel establishment. The tunnel binds the randomized SSH port on the EC2fuzz client to the Samba port running on the localhost network interface of the fuzzing instance.

Once the tunnel has been established, mount_instance generates a mountpoint on the local filesystem by creating a variable `RUN_MOUNTPOINT` which consists of the `BASE_MOUNTPOINT` variable concatenated with a subdirectory titled `RUN_$CURRENT_RUN`, where the `$CURRENT_RUN` variable is substituted with the run number of the current run. The fuzzing instance Samba share is then mounted on the client machine by mounting localhost on the randomized ssh port, which tunnels the remote Samba connection. After the local mountpoint of the fuzzing instance has been established, the function `install_seeds` is

called. This function copies seed files to be fuzzed onto the remote instance. A logs directory is created as well to contain fuzzing run output logs.

After seed files have been copied to the remote instance, the `create_peachpit_template` function is called. Currently this is a placeholder that copies a custom written peachpit peach fuzzer configuration file to the remote instance. In the future it can handle custom peachpit creation for mutation based fuzzing of a binary by substituting the binary into a template peachpit. The peachpit file is then installed with the `install_template` function, which copies the peachpit configuration onto the fuzzing instance. Once the template has been copied, the fuzzing run is initiated using the function `start_fuzzing_run`. The first step in the start of a fuzzing run is the installation of windbg, the Windows debugger released by Microsoft [http://en.wikipedia.org/wiki/WinDbg]. In order to install a standalone version of windbg, I had to manually extract it from the Microsoft SDK package, which means I am unable to bundle it in this software. Currently this installation specifies a path in the local directory to the windbg binary. In order to distribute this code, I cannot distribute the windbg binary so users will be forced to repeat this process. There are comments in EC2fuzz explaining the extraction process from the Windows SDK ISO. Once windbg has been copied to the fuzzing instance, the winrm command is used to remotely initiate the installation in a headless fashion using the Microsoft utility `msiexec` [http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/msiexec.mspx?mfr=true].

Following the installation of windbg, the fuzzing run is launched. The `winrm` command is used to remotely execute the '`Peach.exe`' command, which has been added to the Windows path during the custom AMI configuration. The '`Peach.exe`' command is supplied the path to the uploaded peachpit file ('`C:\fuzzing_run\new_template.xml`'). All standard output and

standard error of the winrm command is redirected to a local file '`fuzzing_log`' on the client and the process is backgrounded so it can run without interrupting the EC2fuzz script. The process id of the command is then stored so runs can be queried and run control can be automated. All fuzzing output is stored to the log directory and is available on the client system through the mountpoint of the fuzzing instance.

Userdata file

The userdata file is borrowed from the masterzen.fr file used in the blog post 'Bootstrapping Windows Servers with Puppet,' which can be found at http://www.masterzen.fr/2014/01/11/bootstrapping-windows-servers-with-puppet/. It is modified by EC2fuzz to write a custom password and custom URL into the file. The userdata file contains Windows Powershell commands to change the password of the instance and download and execute a remote powershell script.

Userdata remote configuration file

The userdata remote configuration file is the powershell script downloaded by the powershell executed in the userdata field during instance boot. It is based on the remote configuration script also taken from the aforementioned blog post and modified to execute tasks necessary to creating an instance configured for fuzzing. The notable modifications include code to download and install Peach fuzzer, code to download and install the WinSSHD SSH daemon, and the port authorization for the SSH daemon.

Analysis

EC2fuzz is a proof of concept tool for automating scalable cloud fuzzing with virtually no prior configuration. The tool is ultimately a Development Operations (DevOps) tool for building a customized cloud configuration tailored to fuzzing and executing fuzzing runs on the configured

infrastructure. It performs both UNIX and Windows automation tasks, and performs automation tasks for Windows from a UNIX environment and implements UNIX network protocols in a Windows environment. EC2fuzz is able to take a stock Windows Server virtual machine and configure it to run additional binaries and programmatically install dependencies. It tunnels firewalled network protocols through SSH to allow for easy access to remote instances on the local filesystem, which makes installing files and collecting logs very simple. EC2fuzz represents a researched solution to difficult cloud automation tasks in multi-platform environments.

EC2fuzz took considerable time to prototype due to the requirement of performing computationally expensive operations as prerequisites to testing code. All remote Windows automation code had to be manually debugged on remote Windows instance after a full run up until that code. All working code had to be refactored to be executed over a network from a UNIX system with remote calls constructed dynamically using environmental variables. Most of the AWS automation and API usage required considerable research and debugging to make sure the proper data was being extracted and this was the case across runs and across states.

Bash is a great prototyping language and reduces development time up until a task scales to a certain degree of complexity. There were a number of tasks that were quite difficult to perform that would have been simpler in a real programming language. However, Bash was effective as a DevOps automation language in that it allows fine grained, well documented use of external tools, which was essential for prototyping a system like this. Powershell is a very interesting addition to the Windows OS that provides quite reasonable automation capabilities. It took me a fair amount of time to become comfortable reading and writing a new scripting language designed for Windows but it was highly valuable to learn.

I actually didn't spend very much time fuzzing. I spent most of time designing a system capable of fuzzing at scale in the cloud with automated provisioning, snapshotting, cloning, and scaling capabilities. At the end of the day, now that this system is in place, it allows me to automate and scale any multi OS tasks that require custom OS configurations for virtual machines.

Related Work

Google ClusterFuzz

Engineers at Google have developed a tool for Chrome security testing called Cluster-Fuzz. Apparently it is a 2000+ core system that fuzzes hundreds of millions of test cases and de-duplicates crashes to provide unique bugs. Google claims that the ClusterFuzz system has helped them find over a thousand bugs in Chrome since 2009. However, other than a few press releases, there is very little information about the actual specifics of this system.

PeachFarmer

iSEC partners, a software security firm, has written a tool called PeachFarmer for log aggregation of Peach fuzzer runs across multiple machines[?]. Peach farmer does not address automated deployment or scaling.

Future Work

I am particularly interested in the Netzob tool [http://www.netzob.org/] for integration into this infrastructure. Netzob has the ability to perform protocol reverse engineering of network protocols and create PeachPit files for undocumented and documented protocols dynamically.

Dead Ends / Paths Not Taken

Without using Windows Server Base AMI on AWS and wanting to fuzz Windows binaries, this process is trickier when attempting to minimize resource consumption and cost. Anybody looking for a Windows XP SP3 AMI will be hardpressed to find a good solution. There are some specialized providers focusing on spinning up legacy browsers for tests of web application across browser suites (http://www.hens-teeth.net/cross-browser-testing/). Other than that the standard solution seems to be: don't use AWS (you can't), revert to vmware cluster. This is undesirable as a solution as the remote infrastructure is part of the value of cloudfuzzing.

The only reasonable sounding approach is to consider something like AWS Workspaces (https://aws.amazon.com/workspaces/details/). However, this Windows virtualization is actually focused on being a fully featured, realtime remote desktop for use as a workstation with respect to virtual corporate presence. This means your laptop is basically just a thin client to a high powered windows machine. As such you are expected to pay for the full licensing of the Windows OS because it is comparable to running a Windows desktop machine on virtual infrastructure connected over a protocol similar in nature to VNC. This results in higher pricing per instance and all remote management being done -- or at least automation is required to be bootstrapped through -- a virtual display adapter with no automation other than mocked clicks at specified coordinates corresponding to graphical prompts. That is absurd.

The next best technique is to go with OpenStack, an open source cloud provider where you have more fine grained control of the initial vm to launch. This, however, requires a private cloud. The next best technique is to go with a vmware or virtualbox cluster on commodity hardware owned or accessible. This allows for custom OS snapshots being supplied for virtualization without any real setbacks.

In the event of being tasked with binary run automation in a vmware cluster, one tool worth considering is cuckoo (http://www.cuckoosandbox.org/). It is a scalable malware analysis sandbox that can run on vmware / virtualbox clusters. It seems reasonable to make a slight modification to the case logic of malware analysis and write fuzzing case logic in its place. This is what I did with cuckoo sandbox. However, it doesn't run on a cloud based system and the porting of the tool from a malware sandbox to a fuzzing system is not simple as the codebase does not separate this logic in a modular manner from general sandboxing techniques. It was partially completed with most of the software not originally by myself (due to it being a software fork).

Bibliography

[1] https://deepsec.net/docs/Slides/2012/DeepSec_2012_Peter_Morgan__John_Villamil_-_The_Whole_Nine_Yards.pdf

[2] http://www.vdalabs.com/exec_mining/Towards_an_Automatic_Exploit_Pipeline.pdf

[3] http://shop.oreilly.com/product/9780596527488.do

[4] http://en.wikipedia.org/wiki/Halting_problem

[5] https://www.immunitysec.com/resources-freesoftware.shtml

[6] http://www.kali.org/news/kali-linux-amazon-ec2-ami/

[7] http://docs.puppetlabs.com/pe/latest/cloudprovisioner_aws.html

[8] http://caca.zoy.org/wiki/zzuf

http://www.masterzen.fr/2014/01/11/bootstrapping-windows-servers-with-puppet/

https://research.microsoft.com/pubs/121494/paper.pdf - Fuzzing in the Cloud

http://technet.microsoft.com/en-gb/sysinternals/bb897443.aspx - Sdelete

http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-resize.html - Resize store backed instance

http://msdn.microsoft.com/en-us/library/aa384426(v=vs.85).aspx - winrm - windows remote management

http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AESDG-chapter-instancedata.html - userdata / instance metadata aws

https://chocolatey.org/ - package manager for windows

http://support.microsoft.com/kb/255905 - how to use orca database editor to edit msi

http://msdn.microsoft.com/en-us/library/aa370557(v=vs.85).aspx - orca documentation

http://dmitrysotnikov.wordpress.com/2008/05/15/powershell-on-server-core/ - powershell on server core / orca hacking

https://gist.github.com/Iristyle/1672426 - ec2 powershell bootstrap

http://www.microsoft.com/en-gb/download/confirmation.aspx?id=11310 - windows sdk and .net for windows server 2008

http://www.darkoperator.com/blog/2013/1/10/powershell-basicsndashthe-environment.html

Other Cloudfuzzers

http://w.rdtsc.net/SOURCE_Dublin_2013.pdf

https://deepsec.net/docs/Slides/2012/DeepSec_2012_Peter_Morgan__John_Villamil_-_The_Whole_Nine_Yards.pdf

Code Bibliography

http://www.howtogeek.com/tips/how-to-extract-zip-files-using-powershell/