Harder, Better, Faster, Stronger: Semi-Auto Vulnerability Research

Richard Johnson
Lurene Grenier

Sourcefire, Inc
July 2010

Much work has been presented in the past few years concerning bug discovery through fuzzing. Everything from the feasibility of exhaustive generation fuzzing, to the continued productivity of simple mutation fuzzing has been covered. This paper will build on this by making the assumption that finding bugs is a foregone conclusion, and will instead discuss the pre- and post-fuzzing process necessary to efficiently analyze vulnerabilities for a given program to the stage where exploitability has a high confidence, and exploitation can be handed off or undertaken in house. This process will be driven by intelligent, analyst-driven automation, with a focus on the continued production of exploitable bugs with a minimum of wasted effort.

1 Introduction

Input fuzzing is a technique in which data is programmatically generated and provided to a program in an effort to exercise available code paths and expose memory corruption flaws. This method of flaw discovery has been proven a necessary step in the testing methodologies of both attackers and defenders. For attackers fuzzing is an essential process because it represents the highest return on investment with regards to time and effort. It is a process which can be undertaken with little startup cost and in parallel with more intensive efforts such as reverse engineering. Further, this fuzzing can be split into a first stage of simple mutation fuzzing while a more in depth generation stage is prepared. This ability of parallelization combined with a high rate of return is unmatched by other techniques.

Due to the above reasons, network defenders as well as software developers can be assured that those targeting their assets will make use of this technique. While code review is a more complete method of bug discovery, it is by no means fool proof. Simple mutation fuzzers often find bugs that are difficult to identify under a manual code review and generation fuzzers are capable of exhaustively testing protocols and file formats, therefore a defender's fuzzing must necessarily be more complete than an attacker's.

While an attacker searching for an inroad into a system need find only a single high-quality bug, and has the benefit of time, a defender must find the great majority of high-value bugs before

release. This will necessitate a multi-tiered solution comprised of speedy mutation fuzzing, more complete generation fuzzing, code review, and regression testing. That said, this paper is not focused on the requirements of a defender.

The focus of this paper is on the goals of an attacker. Specifically we will consider the goals and resources of two specific minded types of attacker. Firstly, we will consider the attacker who has a target in mind for long term and consistent infiltration. We will assume that this attacker has an idea of the software and systems in use by his or her target. Secondly, we are concerned with the freelance researcher whose goal is to reliably discover exploitable vulnerabilities, and weaponize them for sale. These two attackers are linked by common goals and concerns. They each must produce consistently high-value vulnerabilities, while simultaneously conserving human as well as CPU time. Human time and effort must be freed up for the more complex and less robotic task of exploit development, while wasted CPU time directly subtracts from bug yield. Thus, efficiency and consistency are our goals, while reproduction of work and wasted human effort are our enemies.

We will rely upon the prior work of others to convince the reader of the effectiveness of fuzzing itself, and only touch briefly upon the choice of fuzzer as it pertains to the cost-benefit analysis of the process as a whole. We will also assume that the choice of target is outside the control of the attacker, and is instead chosen by environment, or market pressures. We will begin with an overview of the process by which our workflow was created, step by step, before providing example solutions that proved effective for the authors in practice. Next, we will cover the specifics of the automated solutions which were created to facilitate this workflow. Finally, we will contrast the full solution with a more straightforward scenario in which a researcher simply runs a fuzzer and deals directly with its output.

2 Vulnerability Research Workflow

Traditionally, vulnerability research has been focused on bug discovery methods, while the surrounding process is left as an exercise for the reader. Generally, the reader has interpreted the required exercise to simply involve starting the fuzzer and consuming snack foods. This is because the goal has always been to simply find bugs, at which point you provide a collection of several hundred crashes to a vendor and leave the triage work to people who have the code. If, however, your goal is to produce high-value vulnerabilities for discretionary use, a different problem presents itself. This problem does not consist of how to force a fuzzer to produce crashes, but rather what to do when your fuzzer produces an excessive number of crashes of unknown quality.

The standard fuzzer use case includes the following steps:

1) Select a target
2) Select a fuzzer
    a. If the fuzzer is a mutation fuzzer, select input data
    b. If the fuzzer is a generation fuzzer, create the necessary templates
3) Run the fuzzer

4) Evaluate output files by hand for exploitability
    a. 1st pass - remove certainly unexploitable bugs
    b. 2nd pass - select probably exploitable bugs
    c. 3rd pass - select a single bug worth putting significant triage and exploit development time into

The target selection process itself may comprise multiple parts. This paper assumes that the program is chosen largely by market factors and so is outside power of the attacker. However, the issue of attack surface as target decision should not be ignored and will be discussed in a later section. The concentrated fuzzing of independent attack vectors and code paths in general can be beneficial. Further, an attacker may decide to focus on specific areas due to network design or code age.

Apart from this, we should be chiefly interested in improving the second and fourth stages, as they will comprise the bulk of the human effort.

Fuzzer Engine Selection

Fuzzer selection comes down to a choice between the use of human time and the use of CPU time. Use of a generation fuzzer is the more exhaustive option, and will in general find more bugs, however from the standpoint of an attacker this may not be the optimal choice. The reason for this is that a generation fuzzer requires a good deal of time to create the original template for each separate format or protocol you wish to attack. This is tedious, time consuming work which will take time from exploit development, testing, and documentation. More often than not, this expensive human time can be replaced with CPU time in the form of a less intelligent mutation based fuzzer. The question then is how hard is the target; does the time to find a bug with a mutation fuzzer mean that developers will be stuck with downtime, and can this be rectified with more hardware? Note that this trade off can only be made by an attacker; it would be negligent on the part of a defender to ignore generation fuzzing altogether. In the case of an organization with sufficient resources to hire dedicated template writers, generation fuzzing should be undertaken in addition to mutation fuzzing. As Charlie Miller notes, the bug sets discovered by the two fuzzer types often can be disjoint.

Input Set Population

We should point out here that mutation fuzzing is not without its startup costs. The choice of base input is of the utmost relevance to bug count. A poor choice of input file may well result in an empty-handed exploit development team. The primary concern here is that the base input chosen exercises a majority of the code paths concerned with the functionality being tested. Code path coverage should be distinct from simple code block coverage in this measure; this will be covered in section 3. It can be helpful for larger targets to split fuzzing efforts by functionality to be tested. This will allow developers, later in the process, to reuse recently reverse engineered knowledge of the section of the program in question. It will also allow you to quickly populate a stable of inputs and begin testing. Attempting to compile a set of inputs to exercise the entirety of a complex

program can take quite some time. It can also be useful to constrain research to a specific area of a common program in order to reduce the likelihood that others perform research in that area.

One solution for expediting the population of an input set is to simply gather as many files exhibiting a certain trait as possible, then filter them down. For example, if one wished to find a JBIG2 vulnerability in a PDF, they could follow these steps in collecting input files for a mutation fuzzer. Firstly, collect a reasonable sample of PDF files from the internet, possibly with a simple Google search-and-scrape program. Next, you might utilize a code path discovery program to narrow down your large stable to one that exercises the majority of the functionality you wish to test. Section 3 will discuss a possible implementation of this functionality utilizing dynamic binary instrumentation.

Fuzzer Execution

A distinction must be made here between the fuzzing engine, which performs the creation of flawed input, from the testing of that input and its subsequent evaluation of results. While it is the contention of the authors that the fuzzing engine itself matters little in the discovery of bugs, the testing and evaluation matters greatly where time and effort are concerned. The reason for this is that there are many opportunities in this phase of testing to capture data that would otherwise have to be reproduced later in the process.

The decision on what information to capture depends on the time and space necessary to capture it, and the likelihood of its usefulness. Some information, for example the stack trace and instruction on which the program failed is always necessary, and so regardless of cost should be collected. This data is needed to differentiate individual crashes (or "bucket" them) and therefore will always be required. Other data, such as taint analysis, takes more effort to gather and is only needed when confidence of exploitability is higher. Expensive analysis can be performed later in the process and parallelized depending on available CPU resources.

Metadata Collection

The key to effective data collection is debugger integration. When first presented with a fuzzing engine, the immediate impulse is to see if it can find bugs. Once run for a sufficient period of time, the user is presented with a set of inputs which may or may not cause a crash. In many cases the input generation phase is combined with the testing phase. In this case the user is presented with a set of inputs which will cause a crash or fault in a given process. In the second case, work has been done which must be redone. When presented with an input which exercises a bug condition, the first step of triage for a vulnerability developer is to make a quick pass decision as to the exploitability of the bug. Therefore, it is far more efficient to gather all data relevant to this decision which does not significantly slow down the rate of testing at test time, rather than redo the work of reproducing the crash later. In addition to this trade-off analysis, there are a few goals we'd like to achieve with the data we collect.

1) We'd like to determine roughly what class the bug falls into

2) We'd like to have some idea as to the probability of exploitability (just a first pass)
3) We'd like to have enough information to separate bugs into "buckets" as mentioned before.

To some degree, regardless of the cost, requirements 2 and 3 must be satisfied. Luckily, this is generally done with little impact to speed. The decisions made by the authors with regards to what data to collect, along with the methods used are discussed in sections 3 and 5.

Data Storage

As important as the fact that you store necessary data is the way in which you store it. There are a few goals we'd like to achieve in our data storage.

1) We'd like to store information about all crashes indefinitely
2) We'd like to be able to search for a set of crashes with arbitrary characteristics
3) We'd like to be able to change the criteria of our searches, and the data we store as we learn more about differentiation of crashes
4) We'd rather not destroy too many hard drives in the process

Points 1 and 2 are particularly important and tend to suggest that the most efficient and capable way to store ones crashes is with some form of database. The solution utilized by the authors is discussed in Section 5.

Crash Prioritization

Assuming we've done our job in the area of data storage, several options present themselves as far as crash prioritization is concerned. If we are able to store crashes across fuzzing runs, as well as across differing programs, then it becomes feasible to pick targets over time which lend themselves easily to the exploitation techniques of the day. While recent work has endeavored to determine the exploitability of crashes, it seems that currently these solutions provide a suggestion at best. For this reason, it is the contention of the authors that rather than consider the absolute exploitability of a given crash, it is more useful to consider crashes in terms of ease of exploitability. Utilizing the searching capability we've created, it becomes possible to search out and selectively triage bugs with properties that suggest that they may be exploitable with a given technique. As the understanding of what is possible to exploit and what is easily exploitable changes with the release of new techniques the attacker may modify searches to find examples which were once difficult and time consuming but are no longer. In addition, this allows the attacker the ability to make a basic guess as to the effort necessary to produce and exploitable bug.

Depending on one's crash-flow, and the use for which the exploits are being created, it can also be useful to prioritize in a secondary fashion on other data points. For example, one might give preference to readily exploitable crashes for which there are very few examples found in its bucket. This would suggest that the bug in question is less likely to be found and then fixed, ensuring a longer shelf life to the end product.

Crash Reevaluation

Since it is our goal to extend our stable of possibly exploitable crashes across boundaries of both programs and time, it is necessary to perform some form of crash reevaluation as programs are updated. Crash reevaluation is simply the act of retesting the currently known crashing test cases from a prior version on the new version of the software. To accommodate this, it is necessary to store the version number of the target software as crashes are discovered. Remember that crashes occurring only in older, but still in use, software is still quite useful.

Attention paid to changes in software over time can provide valuable insight into the prospective shelf-life of a crash. If a given crash exists in an older version of software while not in a newer version due to changes not related to an explicit bug fix, one may be able to assume that despite reevaluation of the code by developers, the bug was not discovered. In the future, the older, buggy code is not likely to be checked again, and other bug hunters are less likely to search in the old version for bugs when a new version exists for testing.

Performing crash reevaluation and subsequent database updating is a simple and speedy process and will save a good deal of time. If an attacker does not perform crash reevaluation they are left with two time consuming options. The first is to rerun all tests. While it is possible that new bugs could be added and that this would ferret them out, it may or may not be worth the time to check if no significant functionality has been added to the program. If significant functionality has in fact been added, rather than re-run all tests it can be faster to tailor a set of input cases to cover only the new code in question. The other option is to retest buckets by hand when a crash is chosen for further testing only to find that it has been patched.

Bindiff may be used to aid this process. When new versions of the target become available, the attackers may diff the versions and subsequently make use of code path coverage analysis to selectively target changed code.

Workflow Overview

Based on time and perceived target posture the attacker makes an initial decision on fuzzer type. If a mutation fuzzer is chosen, a stable of input files is collected, then narrowed down to a reasonable size based on code path coverage. If a generation fuzzer is chosen, templates are then created for the input formats or protocols in question. Templates, once created, should be tested for completeness through the usage of code path coverage analysis, rather than reliance on the specification. If resources permit, attackers should make use of both.

As crashes are registered within the database evaluations of acceptable bug parameters may be established. Attackers must weigh considerations such as ease of discovery, ease of exploitation, and weight of target software. The considerations which make up this weighted decision are dependent on the desires of the attacker. For example, exploits for immediate use, or low end botnet creation would ignore the ease of discovery aspect. When considering creating a relationship with a high end buyer, ease of exploitability is less important, while ease of discovery is of the greatest importance. With these parameters established, database searches may be created.

If software is updated some automated reverse engineering should be employed to determine the extent of changes and decide if a new stable of inputs should be assembled, or if changes are minor enough to warrant a simple crash reevaluation process.

When database searches have yielded particular buckets of interest, auto-triage work may be undertaken and documentation of the crash can be generated.

3 Enhanced Targeting and Input Selection

The initial phases of fuzzing involve understanding the target program and the input data as thoroughly as possible so as to make informed decisions on where to spend available resources on fuzzing. During this process the program should be analyzed to locate potential attack vectors which include traditional points of interest such as untrusted data entry points or ancillary data such as code age. Once attack vectors have been enumerated, an assessment of which attack should be carried out can be undertaken. This assessment will take into account the fuzzing tools on hand, the available CPU and human resources, as well as the amount of access a successful attack would give the attacker. Once a point of attack has been chosen, the next step is to prepare or acquire the inputs that will be given to the fuzzer. This section will detail enhancements to the targeting and input selection processes.

Attack Surface Analysis

Attack surface refers to the program code that interacts with untrusted data. This data may come from files, registry keys, network packets, and other input sources. The goal is to find the locations where untrusted data enters the program and understand the size and complexity of the code that interacts with the data. This will be accomplished through an enumeration of possible data entry points and call graph analysis.

The attack surface entry points for a program may be enumerated in part by using static analysis to detect I/O related function calls. The static analysis approach does have drawbacks however, as call graph recovery will be incomplete and the detection may miss calls to I/O APIs that are called through wrapper functions. As such, the static analysis approach works best for kernel mode code using the predefined Windows System Call API or in situations where C++ is not in use and wrapper functions are known or a large multi-modular graph can be generated. The key API calls to take note of as taint sources are read/write or send/recv I/O functions. Additional hooks are required to identify the files, sockets, or registry keys associated with the taint:

File	NtOpenFile NtCreateFile
Network	connect listen
Registry	NtOpenKey NtCreateKey
Memory	NtCreateSection NtOpenSection
Process	NtOpenProcess
Thread	NtOpenThread
Event	NtOpenEvent NtCreateEvent

Any functions in the target binary that can reach these entry points on a

program call graph are interesting locations that may be manipulating external data. The file path and permissions associated with these calls will reveal whether an attacker can modify these inputs.

Alternatively, dynamic analysis can offer a more reliable approach to discovering the attack surface exposed during an execution of a program. In the case of mutation fuzzing, dynamic tracing can help determine what attack surface is exposed by a particular input. This approach can also be used to determine whether a selected input exercises an area of code that may be of interest to the attacker, such as an area known to be prone to vulnerabilities or an area of code that has not been modified in a long time.

The solution the authors have chosen currently implements detection of the high-priority attack surface involving file and network I/O using dynamic analysis. Future work will add support for attack surface that less commonly results in vulnerabilities such as registry keys and Windows event objects.

Input Selection

Knowledge of the attack surface allows the attacker to optimize the fuzzing efforts to concentrate only on inputs relevant to the targeted area of code. Specifically, this process requires tracing the execution of the program in order to record basic blocks and API calls relevant to program input. The solution implemented by the authors utilizes PIN, a dynamic binary instrumentation framework, to facilitate efficient program tracing. Trace efficiency is fairly crucial as the process of determining input priority will require a recording of the program execution once for each supplied input. Traditional solutions for execution tracing require the use of the Windows debugging API. Use of the debugging API requires context switching at each block executed which negatively impacts performance. It is because of this performance consideration that the chosen solution utilizes a dynamic binary instrumentation approach.

Once the trace has been recorded, a program execution graph can be built. The recorded trace will include data that records information about loaded modules, block execution, and attack surface related API execution. A graph representation of the program execution can be built based upon the data in the trace. The edges of the graph can be inferred by the order of the blocks and the API trace facilitates filtering of the graph down to the relevant sub-graph that is reachable from the blocks that take input from the target input source. Once constructed, the graphs can automatically be analyzed for differences in coverage and a ranking can be achieved based upon inputs that closely match the targeted fuzzing criteria.

4 Fuzzing Process

The basic fuzzing process is comprised of four steps. These steps are input generation, process setup, testing, and data collection. Input generation should be handled by your fuzzing engine, and as previously mentioned, is the least important aspect of the process.

The first important step is to ensure that the process to be tested is in a state that will facilitate simple and accurate collection of data should a crash

occur. This includes the setup of a debugging environment. Ideally, post mortem debugging is to be avoided in favor of a process opened in a debug environment due to timing issues and changes that are made to the process space in the post-mortem scenario. When running the process in a debugging environment, some knowledge of the process can sometimes be necessary to collect the most useful data from a crash. In many cases, a process will throw, then properly handle exceptions that will cause your debugging environment to trap, but are not useful for exploitation. In these cases, the debugger must be told to ignore the exceptions in question. Other processes may however be unable to handle these exceptions, and ignoring them may cause the data gathered at the time of an unrecoverable crash to be less useful. In addition, the process setup is the proper time to ensure that if there are applicable crash discovery technologies (such as page heap) that they are in place for the new process. The authors' solution was implemented with a custom Windows debugging API wrapper in which the process was set up and launched.

The next step of the process is to provide the input to the program which is being tested. The primary issue of concern here is when a test may be considered completed. The simple solution to this is to observe the process with manually provided inputs and decide upon a static time. This can be both wasteful in some test cases, and too aggressive to catch crashes for others. A better solution is to baseline the process in an idle state, then monitor CPU and memory usage for a return to this baseline once the input is processed. In the authors' debugging wrapper, CPU monitoring is performed with the Windows WBEM interface. A maximum test length may also be provided through a configuration file.

Finally, should the program throw an exception that isn't ignored by the debugger, data collection should begin. The first step here is to collect the necessary information to produce a unique fingerprint for the crash. This will be used in the "bucketing" process to determine if a particular crash was previously seen and what variations exist on the crash. Generally, this can be done by collecting the stack trace and the crashing address, and creating a hash of the data. Next, the entire block of assembly containing the crashing instruction should be collected. This allows for later categorization of the crash as well as a simple indicator of exploitability. Lastly, registers and crash-specific metadata should be collected and stored. The tested solution collected this data using the Windows debugging API, then passed it on to a data storage server with curl where it was then stored in a database for later in the triage process.

5 Assisted Triage Process

Finally, once an input has been generated that results in a program exception, the triage process can begin. The goal of the triage process is to determine the triggering condition, exploitability, and root cause. To accomplish these goals effectively, a final deep trace is required to record the register state and dataflow properties of each instruction. Specifically, all memory addresses that are read or written by an instruction are recorded so

that an additional dataflow graph can be constructed. This graph is analyzed from the dataflow entry points specified by the traced APIs to identify instruction dependencies and propagate taint across instructions that interact with the fuzzed input. The resulting taint graph can be overlaid onto the call graph and control flow graph to enable the researcher to visually identify the flow of execution and tainted data. This final view advances the three steps of the triage as described below.

Exploitability

Exploitability can be determined by identifying tainted data in the context of the crash. The surrounding code should be analyzed to determine which instructions impact program control. If tainted data is referenced by instructions prior to the crash, the attacker has found a condition that is highly likely to be exploitable. If the control instructions prior to the crash are not impacted by tainted data, then static analysis can be performed to determine exploitability. The use of symbolic execution and constraint solving are outside the scope of this paper, however they can be used to determine if future control instructions are influenced by part of the data that is currently under user control at the time of the crash.

Root Cause

Root cause analysis is an important part of vulnerability assessment for both the attacker and the defender. For the defender it is important to identify root cause so that attempts to fix vulnerabilities are made at the original source of the problem. Failure to do so will result in patches that may be bypassed by slightly modifying the input. For the attacker, root cause analysis can reveal the true impact of a vulnerability. In some situations an input will crash in a path that is impossible to fully control, however by analyzing the call stack leading to the crash an alternate code path may reveal itself to be exploitable.

One approach to automating root cause analysis is done through graph analysis. The graphs of two or more crashing program executions using similar inputs should be compared to determine if tainted bytes referenced by the crashing instruction are tainted by the same source bytes. If the crashing instruction differs but the bytes in both crashing instructions are sourced from the same file location then a root cause other than the crashing instruction exists. The root cause may be represented by one of the crashes. This can be determined through manual analysis.

Triggering Conditions

In the case of mutation fuzzing, the triggering conditions are known, however when a foreign crash without knowledge of the input modifications presents itself the triggering condition must be identified to begin the triage process. Triggering conditions can be determined by performing reachability analysis on the taint propagation graph. Reachability analysis finds paths between two nodes in a graph, and as such can be used to determine the code location that originally interacted with user data that lead to the program exception. The current solution takes into account only reachability and does not fully analyze the program constraints on the data.

6 Conclusions

Through the course of research the authors began with a simple fuzzing framework, and worked outward, determining pain points and alleviating them with automation. As work continued it became apparent that the important factors in the continued discovery of exploitable vulnerabilities was not so much the exhaustive nature of the fuzzer, but rather the post processing which separated what are currently considered to be exploitable bugs from those which are certain to never be exploitable, and also from those which may be exploitable with a good deal of work.

While the decisions made by the authors in their own work are specific to their goals in fuzzing, they have done their best to enumerate the considerations taken into account when coming to these decisions and list them for the reader.