Editors: Michael Howard, mikehow@microsoft.com
James Whittaker, jw@se.fit.edu

Violating Assumptions with Fuzzing

Fuzzing (*fŭz'ɪŋ*) *n.*—a highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. From modem applications' tendency to fail due to random input caused by line noise on "fuzzy" telephone lines.

Peter Oehlert
Microsoft

Fuzzing lets developers or quality assurance (QA) teams test large numbers of boundary cases when doing so with techniques such as functional testing would be cost prohibitive. Comprehensive negative test cases—those that verify that a product does not do something it shouldn't do, rather than that it does something it is supposed to (positive test cases)—are difficult to construct because the number of possible permutations is astronomical. Yet, fuzzing covers a significant portion of negative test cases without forcing the tester to deal with each specific test case for a given boundary condition. If your input includes a 4-byte signed integer that should be between 1 and 10, for example, boundary cases would include 0, −1, 11, 12, large negative numbers, and cases around the byte boundaries (2^8, 2^{16}, 2^{24}, 2^{31}). Coding these cases individually would be difficult, and this is really a best case set of test cases given the fact that an integer is a relatively restricted input form (strings have far more permutations). Boundary conditions are important because significant subsets of boundary condition failures are security failures. As such, the boundary conditions we don't test today are the security patches we'll have to issue tomorrow.

A fuzzer tool generates *semivalid* data (data that is correct enough to keep parsers from immediately dismissing it, but still invalid enough to cause problems), sends it to a target application for processing, and then observes the application to see if it fails as it consumes the data. If so, the tool saves the submitted data for later analysis and submits new malformed data. If the application doesn't fail, the tool chooses whether to delete the malformed data and continues to the next iteration. Performing these steps manually, we could perform only several hundred or perhaps a few thousand iterations. By automating the entire cycle, however, a fuzzer tool can perform hundreds of thousands or millions of such iterations, covering a significant number of interesting permutations for which it would be difficult to write individual test cases.

When writing secure applications that take input from untrusted sources, developers must test input parsers for all manner of boundary conditions. Fuzzing can make this process significantly easier and provide the best results for the time allocated to testing, helping to uncover issues in data parsing that can otherwise remain unnoticed. For example, Microsoft shipped the HyperTerm program—a terminal emulation program—with Windows for many years. It was written in a different time when security wasn't as much of a concern as it is today, and although the company reviewed it, like all legacy code, during the Windows Security Push, the .ht files weren't considered vulnerable to potentially malicious input; consequently, testers didn't investigate the file parser at that time. Moreover, the .ht extension is associated with HyperTerm by default, which means a mail client or Web browser could automatically pass a malicious .ht file to a vulnerable program if opened as an attachment.

Someone outside of Microsoft recently found a security issue somewhere in the .ht format that required a security bulletin. Using a fuzzer uncovered eight other exploitable security issues in HyperTerm's file parser within the first few hours. Further fuzzing over the next few days produced four additional vulnerabilities. This testing technique, combined with a detailed security code review, led us to fix additional issues in the program. One of the best uses of time for the test resources during the investigation was to create a fuzzer for the .ht file input.

Methodology

Fuzzing involves writing a tool that will generate semivalid data, submit it to an application, and determine whether the application fails. For the purposes of this article, I will consider only automated approaches to fuzzing. Figure 1 illustrates the different high-level states a fuzzing tool moves through.

A *trust boundary* is any place that data or execution goes from one trust level to another, where a trust level is a set of permissions to resources. Transitioning from the user mode in an operating system to the kernel crosses a trust boundary, for example, because the kernel is trusted to do anything with the processor, whereas the user mode allows only a subset of processor operations. Similarly, a trust boundary exists between a network and a machine because anyone on the network can modify network data, whereas only someone on the machine can modify the machine data. Crossing between different user contexts presents yet another trust boundary. Another way to look at trust boundaries is as locations where vulnerabilities result in the elevation of privileges. We must consider trust boundaries when deciding what to fuzz in an application; they help prioritize which inputs to look at and the order in which to do so.

Applications often have multiple inputs, but if the product team does a good job with threat modeling, the threat model will contain a detailed list of them all. In most systems, the majority of input comes from files, config and registry entries, APIs, user interfaces, network interfaces, database entries, and command line arguments. These inputs are therefore prime targets for fuzzing, but anyplace that a system receives input is a candidate for submitting malformed data.

The next step is to prioritize which of those inputs to fuzz, which can be a tricky process. For example, I recently reviewed a Windows service with a network interface that took data from an admin-only authenticated connection. It also read a registry key through which the user could specify which folders were important to search. The development team thought it would be important to fuzz the network interface because the component was network facing. Instead, I recommended that they focus on the registry entry because any authenticated user could write to the registry key. This input to the system crossed a trust boundary and represented an elevation of privilege from any user to the service account. The network interface could be attacked only by users who were already administrators because it didn't cross a trust boundary; in contrast, a vulnerability in the software meant anyone who could log on to the machine or access the registry remotely could write to the registry setting that let them elevate to the service account.

Once you decide which entry points to fuzz, many different techniques are possible, but the fuzzer first needs a source of malformed data. Note that if all the submitted data were malformed, the application would throw out the input after parsing the first invalid data block, and none of the other code would be tested. Thus, it's critical to use data that is mostly valid but includes a few invalid or semivalid pieces. There are a two main ways to get this data: *data generation* and *data mutation*.

The fuzzer can generate data based on a specification for how it should look. The data description could be as simple as "it's an int." The actual description will depend on the language your application is written in, but it should not be ambiguous for your specific application. (I'll ignore special complex cases such as network applications, which might require integer reordering based on whether the integer is written or read from the network or local machine; in most cases, an integer is the simplest case.) Alternatively, the description might

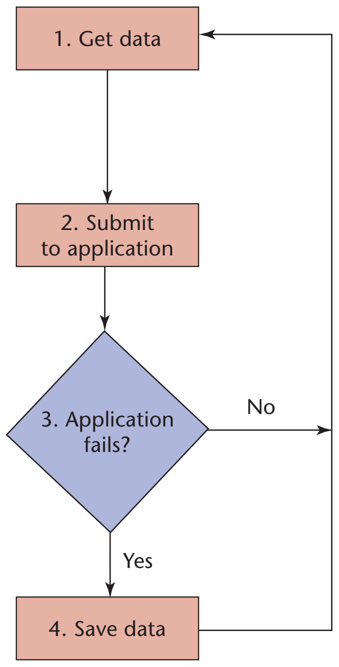

Figure 1. A complete fuzzer iteration, starting from generation. The fuzzer begins by getting semivalid data via one of the two main methods for use in testing: generation or mutation. The fuzzer then submits the data and tracks whether the erroneous input causes the application to crash (in which case, it saves the data for later analysis). If not, the fuzzer automatically proceeds to the next iteration.

be as complex as an XML document that describes individual offsets and data structures for an arbitrarily complex nested binary data structure. Of course, this would make implementing the fuzzer more difficult because we first have to understand the format's specifics and then create an XML schema to accurately describe the format before finally writing a fuzzer to parse the XML schema and XML document to generate the semivalid data.

The second way to get malformed data is to start with a known set of good data and mutate it in specific places. HTML is a good example of a complex file format for which it would be difficult to create a generator that could cover the entire specification. Rather than generating the code from the HTML format specification, the fuzzer can use a valid example or template file, copy and modify it in a few key areas, and submit it to the target application—a Web server or browser. Creating new test cases is as simple as gathering template files from an available source of existing files (in this case, the Internet).

When the fuzzer or someone running it can easily obtain good data sources—for config and registry settings, user interfaces, command line interfaces, and database interfaces, for example—mutation is often the simpler method. In other cases, getting a good copy of the input can be more difficult than simply generating it from scratch (with APIs, for instance). Formats can also be highly sensitive to malformed data. For example, network protocols are often strictly defined so that too much variation, particularly in key control fields, can terminate the parsing early in the code path—perhaps in the network stack before the application sees it. In such cases, mutation ends up being more work because parsing good data to determine what can be altered for fuzzing is difficult.

Intelligent vs. unintelligent fuzzers

In cases like the network example, the fuzzer requires extensive knowledge of the format it is attacking. In other situations, the fuzzer can just randomly mutate some data and submit it to an application with no a priori knowledge of the format. So which technique is better? On one hand, an intelligent fuzzer knows the format, which enables it to specify valid data for most of what it's attempting to fuzz. Intuitively, we can see that an intelligent fuzzer could potentially proceed farther along a parser's code paths and get better code coverage.

On the other hand, an unintelligent fuzzer has no built-in assumptions about the format or application it is attacking; it simply changes bits randomly to see what happens when the target application consumes the data. From this perspective, it's convincing to think such an approach could cover cases far outside the bounds of what the application expects, simply because a given modification seems so unlikely to cause problems that no one has ever tried it. Random changes can expose problems such as those related to null characters in text-based formats, which use text-based control codes like carriage return and line feed (`CR` and `LF`) or special characters (such as `<` and `>`) as delimiters. Nulls are often completely invalid, but applications sometimes treat them as the end of a string.

Clearly, both techniques have some benefits. Investigating the issue a bit deeper, we find a whole spectrum of intelligence we can build into a fuzzer. At one end, the tool might randomly choose a location in a bit stream to insert, delete, or change values; at the other extreme, the tool might be able to exploit its knowledge of a complex format's exact usage and structure. In between are fuzzers such as those that look for simple binary patterns (often problematic in parsers) or those that can specify large portions of a network control protocol but ignore binary opaque blocks in the middle of a packet. Certain applications will dictate the use of one technique over the other.

Patterns

When writing a fuzzer, you'll often find that patterns provide the greatest benefit for the cost. The results of pattern-based fuzzing vary according to the data format's complexity and use of structures and techniques such as checksums or numerous relative referencing fields. Nonetheless, pattern-based fuzzers generally cost less to write than intelligent fuzzers, while providing results of a similar caliber.

A pattern-based fuzzer looks for particular data patterns and then performs some data modification when it locates them. For example, a pattern of *n* number of sequential byte values that correspond to ASCII data in the printable characters range (0x20 – 0x7F) in a binary code block might indicate a string. Similarly, byte values alternating between a value in the ASCII range and zero might indicate unicode data. After identifying this pattern, a fuzzer could take some interesting action, such as inserting additional valid string data into the block in an attempt to find a buffer overrun. As I mentioned, removing the trailing null termination character from an identified string is another useful fuzzing technique because a parser will sometimes expect this character and fail when it's not there. String data structures also commonly prepend strings with their lengths, which means modifying the value immediately preceding a string can expose interesting bugs. A buffer overrun might ensue, for example, because the parser is depending on the prepended length to allocate the buffer and a null terminator to copy the data.

Text formats offer similar targets to string data. The difference is that all data in a text-based format is, at some point, string data. Tokenizing the string data and manipulating it based on the tokens can thus lead to some interesting results. For example, changing the order, inserting duplicates, and modifying the tokens can cause a parser to fail. Additionally, parsers identify delimiters and use them to separate data into tokens. Inserting delimiters in the middle of these tokens or between them can also work well for fuzzing. For text-based formats, using null characters, spaces, `CRLF`, `CR`, `LF`, and encoding sequences can expose parsers' potentially inaccurate assumptions about where these should be.

Integers are a great target with binary data, which often relies heavily on them to specify sizes and counts of arbitrary structures. Simply re-

Glossary

The following terms represent key concepts for fuzzing and other types of code testing.

Attack surface—a general term for the set of code to which an attacker has access. A common measure is bugs per thousand lines of code (KLOC): by reducing code on a system, you reduce the amount of code that could have an exploitable condition.

Code coverage—a technique for determining which pieces of code have run during a specific time period, often by instrumenting the code to be tested with special instructions that run at the beginning and end of a function. When the code runs, the special instructions record which functions were executed. Recording which code a test case covers can indicate how much additional testing is needed to cover all the code.

Format—in the current context, the valid input structure expected by an application being fuzzed. The format for network input will likely be the network protocol, for example, whereas the format for command line parameters will likely be several space-separated parameters with the – or / indicated control parameters.

Invalid data—data that the infrastructure being used to transport it is likely to reject before it can reach the application that consumes it.

Parser—a set of code that takes data as input and converts it to a form that a program can use. This often means deserializing binary or text data formats into structures or classes that a program can interpret easier.

Semivalid data—a detailed document describing an application and its vulnerabilities. Testers use threat models to find architectural security holes and prioritize components for security code reviews and security testing.

placing an integer value with a special value can often cause a parser to crash in strange and wonderful ways. Inserting all 0s (0x00, 0x0000, 0x00000000, and so on) or all 1s (0xFF, 0xFFFF, 0xFFFFFFFF, and such) into a binary block can be effective in identifying integer overruns where an integer is used to allocate memory or index a buffer during parsing. Adding or subtracting some small value from the current value can also locate issues because binary structures are often identified by some type that might be an enumeration with a particular small range. For example, Microsoft SQL Server's Tabular Data Stream (TDS) network protocol uses a byte at offset 4, which must be of the range 0–5, to signify the TDS packet type. A small change to this number leads to a different parsing code path, which might expect a similar but different format, whereas a large change might get the case thrown out by a range check. Flipping the top bit of an integer value to make it negative (bitwise `OR` with 0x80, 0x8000, and so on) can help identify signed/unsigned mismatch errors. Finally, moving data around by exchanging different blocks within a data stream can also uncover bugs.

Common fuzzing problems

When building a fuzzer, you must account for several common problems. Some you can anticipate by closely examining the format you are fuzzing, but you'll discover others after you've begun developing the fuzzer. With a well-understood format, you can probably determine whether you'll have to deal with any of the following validation problems, which can hamper a fuzzer's effectiveness.

Many formats and protocols perform various types of validation. For example, network protocols and file formats often use hashes and checksums to help verify the integrity of packet and file content. Of course, these mechanisms present an obstacle for fuzzing because we need to be able to change the content for testing purposes. The solution is to provide additional logic in the fuzzer to recalculate the correct hash after content mutation or generation.

Encrypted hashes and digital signatures are even more problematic because they're designed to verify not only that data content hasn't changed since the source sent it but also that the source has some known identity. In these cases, the fuzzer also needs knowledge of the digital signature algorithm and the private key used to sign it, so that it can pretend that it is the source party.

With digitally signed or encrypted data, the input represents a significant threat only if the encryption or signed data crosses a trust boundary. For example, encrypted (and thus signed) email is important to fuzz because, even though the sender is authenticated with a public key, the user reading the email might not consider the sender trusted. The encrypted and signed data crosses a trust boundary between the sender and receiver whose trust levels aren't necessarily the same. On the other hand, a patch is an example of data that doesn't generally cross a trust boundary. If a software application validates the software company's digital signature on the patch, it might not be important to consider malicious input because the company is trusted to always provide good data. Of course, if the threat is that a rogue employee of the software company might produce a malicious patch, fuzzing becomes important again. This digression illustrates the value of threat modeling: once you understand the threats you need to protect against, it's much easier to mitigate them by doing fuzz testing against input.

In general, encrypted formats don't lend themselves well to fuzzing. If the fuzzer does anything beyond random bit flipping, its pattern recognizer or parser will be unable to parse the data. The key here is to ensure that the fuzzer has the additional capability to decrypt data before it mutates and reencrypts it; with data generation, it needs to be able to encrypt before submitting to the application. Formats that use compression present nearly identical issues, and the solution is the same: provide for decompression and compression capabilities in the fuzzer.

Of course, this is hardly a comprehensive list of problems you might encounter while developing a fuzzer. Once you have a working tool, you can identify additional problems through code coverage analysis. By running the fuzzer against an application while monitoring for code coverage, you can quickly determine whether all your cycles are being rejected in the same place in the code. By analyzing the specific section of code, you can then determine what to do to let the fuzzer proceed beyond that point (at least some of the time).

Application behavior

What actually happens when you submit fuzzed data to a target application depends largely on what you're fuzzing. If you're testing a network stack, the data travels to the application via the network, but it can be difficult to tell if the data has had the desired effect once it arrives. Determining when an application fails is much easier than proving when it succeeds.

In fuzzing, it's best to start by looking for the things that are really wrong. For example, the Windows operating system uses exception handling to signal failure cases to an application and to other parts of the OS. A debugger can see these exceptions, so building one in to a fuzzer allows it to determine when an application crashes.

Other ways to check application correctness during fuzzing include looking for spikes in the program's memory usage or CPU utilization, which could indicate that the application is using the malformed input to make calculations for memory allocation. It could also indicate an integer overrun condition if the program performs arithmetic operations on the input values. This might result in an exploitable buffer overrun condition. Similarly, a CPU spike generally means that the program is using an intensive algorithm that isn't properly bounded—perhaps as simple as a loop in which the variable used to determine how many loops to perform comes from the malicious input. At the least, this indicates a denial of service if not something more dangerous.

From these simple failure models, we can envision more complex and complete failure and success models that actually examine the system to ensure that it is working correctly after parsing malformed data. I recommend using an extensible pattern in the fuzzer to implement checks for the success or failure of applications parsing the malformed data. This is true particularly because what constitutes success and failure will change over the fuzzer's lifetime.

When creating software, developers must seriously consider how to cover all boundary conditions—focusing particularly on those that can lead to security vulnerabilities. The ideas I've presented here should provide some understanding of how to proceed in building a fuzzer and what to expect from the fuzzing process. □

Peter Oehlert *is a security software engineer at Microsoft. His research interests include finding security vulnerabilities in applications, networks, and databases. He is a Microsoft certified systems engineer (MCSE), Microsoft certified solution developer (MCSD), and a certified information systems security professional (CISSP). Contact him at peteoe@microsoft.com.*