



Codonomicon whitepaper:

How to Really Avoid Zero-Day Attacks – Build Security In, Don't Add it



- 1** Introduction
- 2** Fix Zero-Day vulnerabilities proactively
- 3** Fuzzing and Zero-Day vulnerabilities
- 4** Fuzzing and Vulnerability Scanning
- 5** Fuzzing and other complementary systems
- 6** Why it is better to build in security than to add it

CODENOMICON Ltd. | info@codenomicon.com | www.codenomicon.com

Tutkijantie 4E | FIN-90570 OULU | FINLAND | +358 424 7431
10670 North Tantau Avenue | Cupertino, CA 95014 | UNITED STATES | +1 408 252 4000

PREEMPTIVE SECURITY AND ROBUSTNESS TESTING SOLUTIONS

1 Introduction

Hardening new technologies with traditional vulnerability auditing methods is becoming increasingly challenging. Fast software release cycles and greater complexity coined with decreasing control over the code creation process due to use of third party code and outsourcing increase the likelihood of new and unique security vulnerabilities, wrong configurations and unwanted backdoors in software. At the same time, discovering new bugs is becoming a growing problem. More and more vulnerabilities are never disclosed publicly. Instead, they are sold and distributed within underground hacker communities.

As a result, software development companies can no longer rely on the user community to find bugs. Instead, they have to take a more active role in ensuring the security of their products and services. Numerous companies have started to integrate software security practices into their Software Development Lifecycle (SDLC), an approach, which promotes the idea of building security into systems rather than using reactive post-deployment security add-ons. While, traditional security testing has been overwhelmed by these challenges, fuzzing has proven itself as an easily adaptable testing technique.

2 Fix Zero-Day vulnerabilities proactively

Zero-Day vulnerabilities are unknown vulnerabilities hidden in software code. In contrast to known disclosed flaws with available patches and updates, vendors are unaware of the existence of these unknown vulnerabilities, and therefore they are not prepared to provide fixes for them. The term zero-day comes from the “Warez scene”, a subculture specializing in distributing pirated content: zero-day content is the most valuable pirated content you can have, that is unpublished content, which nobody else has. Similarly, a zero-day bug is a vulnerability nobody else knows off, or at least the vendor in question is unaware of it. And because none of the security vendors know about them either, attacks against zero-day vulnerabilities cannot be detected or blocked by any security defenses.

All unpatched software flaws are potential security threats. Bugs are weaknesses in the system, which can be exploited to attack a system or an application. To find zero-day vulnerabilities, the attackers send abnormal data feeds to a system until they get an abnormal response from the system. After the initial discovery of a potential vulnerability, they continue to edit their anomalous inputs until they get the system to behave the way they want. However, zero-day bugs do not always need to be exploited for criminal purposes to cause problems. Sometimes bugs can be triggered by simple unexpected inputs in events like heavier than normal use, system maintenance, or even natural events like thunder.

A zero-day vulnerability is a ticking time bomb, it does not really matter what eventually sets it off. The important thing is to get rid of them.

The term “window of vulnerability” is often used to refer to the time the system is vulnerable due to a flaw. There are different interpretations of how this time should be measured. According to a purist view, the system is vulnerable from the moment the flaw is made to the moment when all users have implemented the patch. The window of vulnerability can also be seen as the time it takes the vendor to create a patch for the flaw, after it has been reported to them. However, despite these differences, it is always true that the earlier a vulnerability is discovered, the easier and the cheaper it is to fix the flaw, and the more comprehensive the fix is. The best way to avoid vulnerabilities is to write good code, and to ensure the quality of your code you need extensive testing.

3 Fuzzing and Zero-Day vulnerabilities

Fuzzing is a software testing technique, in which unexpected data is generated and fed to the inputs of a system, and the behavior of the system is then monitored. If the system fails, e.g., by crashing or by failing built in code assertions, then there is a bug in the software. Fuzzing is a very representative method to test software robustness, because it targets the problems attackers would also find. The feeds used in fuzzing can either be random or they can be designed to target specific problems. If random feeds are produced, e.g., by

mutating valid inputs, the amount of test cases needed to test the system comprehensively is infinite. The amount of test cases needed can be significantly reduced by targeting the tests, thus making the whole testing process quicker and more cost efficient. This type of intelligent model-based fuzzer is the state-of-the-art in fuzz testing today.

To design intelligent tests, you first need to define the weak points of the system you are testing and then create test cases, which specifically target these weaknesses. Because protocols are the underlying languages systems use to communicate, it does not matter how secure your systems are otherwise, if your protocol implementations contain vulnerabilities. Therefore, it makes sense to target protocol implementations with fuzz tests. Another vulnerable area is file formats. Different file formats are defined by conventions. If these are broken, a system might not be able to process the file. In the worst case, this might affect the operability of the whole system. For example, the process can crash when the file is being opened or the data structures can get corrupted resulting in a loss of data.

There are infinite ways to manipulate protocol feeds and file formats, e.g., input data can be altered by adding and removing characters, by adding special characters, or by repeating elements. You need experience to know what types of feeds trigger abnormal behavior in the tested systems. Such knowledge can be packaged into a test case library, and if the gathered data is comprehensive, the library can be used to test different systems systematically. Such a library could also be used by inexperienced users, because they can use available information, instead of having to define the test cases themselves. For example, the users might only need to know the protocol their system is using to find the appropriate test cases for their system from the library. Such model-based tests also cover implementations, which are rarely used. Malicious input in rarely used implementations tend to cause havoc in systems, because they are not subject to rigorous testing or heavy day-to-day usage.

4 Fuzzing and Vulnerability Scanning

Most traditional vulnerability auditing solutions focus on scanning the systems for previously known bugs. This technique is based on sending requests to a software system and then analyzing the system based on the responses. The benefit of such testing methods is that they do not actually trigger any vulnerabilities, and therefore they are usually safe to use even in production networks. The most aggressive security scanners do contain libraries of known exploits for triggering known vulnerabilities in commonly

used software components, but these are mostly harmless. The biggest weakness in security scanners is that they cannot find previously unknown vulnerabilities. This is especially problematic with emerging technologies and proprietary protocol extensions: there are no ready bug lists, and thus it is impossible to find weaknesses using vulnerability or security scanners.

A zero-day vulnerability is a ticking time bomb, it does not really matter what eventually sets it off.

Fuzzing achieves remarkable efficiency in finding both known and unknown vulnerabilities. Indeed some vulnerability scanners employ fuzzing to test for variations of known bugs. This provides better test coverage than using individual tests to verify known vulnerabilities, but is extremely unlikely that such tests will reveal any new weaknesses. The main strength of fuzzing is its ability to find completely new vulnerabilities from a tested system. The reason for this is that in fuzzing test cases are not based on known vulnerabilities. Instead, they generated using models, which are based on protocol and file format specifications. An additional benefit of using such model-based fuzzing tools is that no signature updates are needed to keep the tests up to date.

5 Fuzzing and other complementary systems

To guarantee the quality of your code, you should integrate security testing into your software development process and employ more than one security testing technique, because not all vulnerabilities can be found using the same method. Fuzzing and traditional methods like vulnerability scanners and code audits should be seen as complementary methods. To perform a complete security audit on a piece of software, you must perform fuzzing, code auditing, load testing, performance testing and vulnerability scanning. Code auditing tools look for code patterns, which are known to be vulnerable, therefore is useful for the auditor to have access to the source code. Load and performance testing are used to test the system for Distributed Denial of Service (DDoS) attacks. A large volume of valid protocol traffic is sent to the system under test to ensure that it is able to handle a predefined amount of traffic, and to find the limits of how much protocol traffic the system is able to handle. Vulnerability scanning tests the underlying operating system and common services such as databases for known security vulnerabilities that have already been reported in the public. However, fuzzing provides important features, which other security solutions lack. In fuzzing, there are no false positives, and moreover, fuzzing is widely applicable. Some testing techniques are only useful for programmers, while others are only practical for system testing. Fuzzing, on the other hand, can be used at every step of the software development lifecycle (SDLC).

6 Why it is better to build in security than to add it

The best way to guarantee security is to ensure that your code is well written. There is no use in trying to protect an impaired system with firewalls and anti-virus software. These merely add to the complexity of the system, and complexity is always a threat, because it increases the attack surface of the system. In fact, these protective devices are actually a major threat. Just think about it: anti-virus programs are installed on all company computers or at least where valuable content is stored, they have the highest authorization and can access all files, and at the same time there are very few vendors for anti-virus software.

By discovering a single flaw in just one anti-virus software solution or firewall, attackers can hit a staggering number of applications. On the other hand, application service providers, like online-banks and web stores, cannot build fortresses around their application, because their customers need to be able to use their services. Problematically, most attacks are made through these public interfaces, because it is the easiest way.

All bugs are potential security threats. The best way to guarantee security is to ensure that your code is well written.

Fuzzing enables testers to accurately simulate potential attacks, and to patch the found vulnerabilities, before somebody else finds them and exploits them. Fuzzers do what attackers do, but before them. By integrating fuzz testing into your software development process, you can discover flaws at the earliest possible moment. The earlier the bugs are discovered the cheaper and easier it is to fix them. You can actually gain time and save money through preemptive security and robustness testing. Instead of, racing against the clock to get your system to work again and spending valuable resources

in calming down angry customers, you can discover flaws and create patches for them proactively, before any problems occur. Furthermore, if vulnerabilities are fixed early on during the software development process, then there simply will not be any time for

anyone to devise an attack. You can also use Fuzzing as an auditing tool to verify the quality of outsourced code development. The costs of bad Quality of Service (QoS) and downtime can be considerable to your company reputation and sales. By ensuring the quality of your software you can also avoid other liability problems. And, most importantly, by building the security into your system, you can be sure that it is really secure and not just coated with security. Remember, if it is a bug, fix it, don't try to hide it.



CODENOMICON Ltd. | info@codenomicon.com | www.codenomicon.com

Tutkijantie 4E | FIN-90570 OULU | FINLAND | +358 424 7431
10670 North Tantau Avenue | Cupertino, CA 95014 | UNITED STATES | +1 408 252 4000