

Codenomicon whitepaper:

Building Secure Software using Fuzzing and Static Code Analysis

- Anna-Maija Juuso and Ari Takanen -



- 1 Introduction
- 2 Building Security into Systems
- 3 Differences in Test Coverage
- 4 False Positives and Negatives
- 5 White-box vs. Black-box Testing
- 6 Static vs. Dynamic
- 7 Vulnerability Management
- 8 Fuzzing and Static Code Analysis in the SDL and BSIMM Models
- 9 Combining Fuzzing and Static Analysis
- 10 Conclusion

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

Tutkijantie 4E | FIN-90590 OULU | FINLAND | +358 424 7431
10670 North Tantau Avenue | Cupertino, CA 95014 | UNITED STATES | +1 408 252 4000

1 Introduction

The increased complexity of new technologies and faster software release cycles are making traditional reactive security solutions ineffective. Instead, more adaptable, preemptive product security testing is needed. Moreover, for example, due to agile development and outsourcing, the software development process itself is also becoming more complicated further increasing the need for more effective product security practices. The Building Security In Maturity Model (BSIMM) and Microsoft's Security Development Lifecycle (SDL) concept combine preemptive security testing with the SDLC models. Fuzzing and static code analysis are both a natural part of these secure development best practices, because they promote building security into systems proactively, instead protecting vulnerable systems and reacting to security issues. In this whitepaper, we describe how fuzzing and static code analysis can be used as complementary methods to ensure the security and robustness of your software.

2 Building Security into Systems

Vulnerabilities Enable Attacks

Vulnerabilities are not created, when a system is being attacked; they enable attacks. Vulnerabilities are implementation errors that are introduced into the code, when the programmer makes a mistake. They become vulnerabilities, once the software is released exposing the software for attacks. Security researchers, security companies and hackers discover some of the vulnerabilities, and if they choose to report the findings, they can enable software developers to create patches for the found vulnerabilities. However, other vulnerabilities still exist in the code waiting to be exposed. These unknown vulnerabilities are called zero-day vulnerabilities. Software vendors are unaware of their existence, thus there are no ready patches for them. Once a zero-day vulnerability is triggered, the developers race against the clock to get it fixed, before irrevocable damage is done to the company's sales or reputation.

All Vulnerabilities are Exploitable

Attackers need to find vulnerabilities in a device or a system in order to devise an attack against it. Basically, any crash-level bug can be exploited to attack a system or an application. Attackers send unexpected inputs to a system, and if they can get an abnormal response from the system, they continue to refine their inputs until they get the system to behave the way they want. Sometimes bugs can be exposed by simple individual inputs, and sometimes attackers have to communicate with longer message sequences with the system, in order to gain access to the deeper protocol layers. In some cases, vulnerabilities can even be triggered by events like heavier than normal use or maintenance. Therefore, the best way to ensure the security of your systems is to build security into it by finding and fixing these critical bugs before they become security vulnerabilities. The goal of both fuzzing and static code analysis is to produce better quality software by finding vulnerabilities in the code before deployment.

3 Differences in Test Coverage

Testing Entire Systems with Static Code Analysis

Static code analysis tools identify problematic programming practices and risky code structures. The purpose of the code review is to ensure that secure coding policies are being followed. Static code analysis is an efficient process for identifying many common coding problems. It examines the entire source code looking for syntactic matches of common coding problems using patterns and rules. The more advanced tools adopt a multi-layered approach to source code analysis, combining syntactical and semantic analysis with other forms of analysis. These include searching for architectural clusters or spaghetti code, analyzing dependencies and coupling, threading issues and metrics. All of these can relate to the security of a system. Figure 1a illustrates that static analysis tools can find bugs in the entire software, but not all the issues it finds are critical or exploitable.

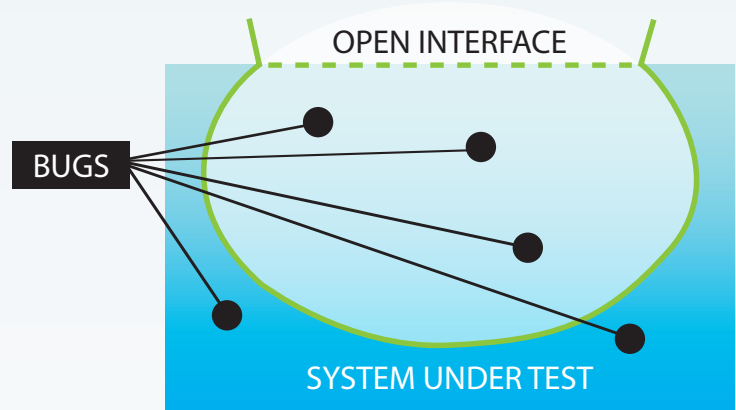


Figure 1a. Static Code Analysis Coverage

Testing Open Interfaces by Fuzzing

Fuzzing, on the other hand, is a form of attack simulation, in which unexpected data is fed to the system through an open interface, and the behavior of the system is then monitored. If the system fails, for example, by crashing or by failing built in code assertions, then there is a bug in the software. Similar to advanced code analysis tools, more advanced fuzzing tools also test multiple layers. It is important that the fuzzers can genuinely interoperate with the tested system. Only this way can they access the deeper protocol layers and test the system more thoroughly. In contrast to static analysis, fuzzing can only find bugs, which can be accessed through an open interface, as illustrated by Figure 1b. However, all the found issues are critical and exploitable.

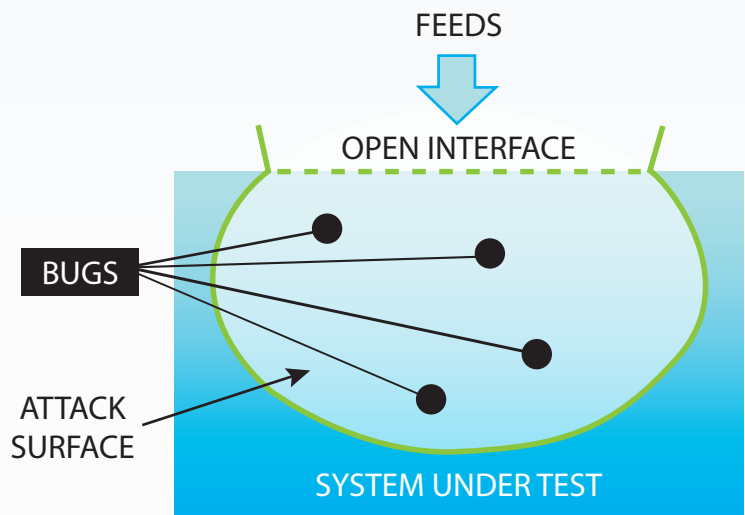


Figure 1b. Fuzzing Coverage

4 False Positives and Negatives

The problem with static analysis is that it also reports bugs that the software does not contain, namely false positives. Because it is impossible to test the entire system, test tools have to make a number of approximations, which leads to inaccuracies. False positives are troublesome, because checking them consumes valuable resources. Thus, some tools have adopted heuristic or statistic approaches. The advantage of these approaches is that they can reduce the amount of false positives without compromising test coverage. In effect, it can improve a tools ability to catch critical vulnerabilities.

In fuzzing, there are no false positives: Every bug is discovered as a result of a simulated attack. Thus, they are all real security threats. Simple fuzzing techniques rely on random mutation to create test cases. The coverage of mutation based Fuzzers is limited and thus they only find some of the vulnerabilities the software contains. Yet, this is better than performing no fuzzing at all, because these are the vulnerabilities the attackers would also find. More sophisticated fuzzing techniques improve coverage by using protocol specifications to target protocol areas most susceptible to vulnerabilities. This type of optimized test generation also reduces the amount of test cases needed without compromising test coverage.

5 White-box vs. Black-box Testing

Performing static code analysis on software requires source code access. Thus static code analysis is essentially a solution for sell-side companies producing their own software, whereas fuzzing can be black-box, grey-box or white-box testing. Therefore, it not only provides solutions for companies developing software, but also for companies, which outsource their code development. Companies with their own software development processes can get the best advantage from security testing by integrating fuzzing and source code analysis into their software development process. Thus, they can find vulnerabilities at the earliest possible moment. The earlier vulnerabilities are found, the easier and cheaper it is to fix them.

Fuzzing is also a quick verification method for third-party developed code. Companies with outsourced code developments can integrate fuzzing into their sourcing processes and mitigate risks by checking their systems before deployment. This type of testing can also take the form of interoperability testing. Moreover, by fuzzing their outsourced software components, the companies can ensure that the price matches the quality and also provides them with tools for price negotiation. In addition, fuzzing can be used by end-users of the software without any access to source code, providing an automated means of quality assurance for later steps in product life-cycle.

6 Static vs. Dynamic

Static code analysis is a good method for improving the general software quality level. A major difference between static code analysis and fuzzing is the intrusiveness of the testing method. Static code analysis can only be performed off-line. As its name suggests, static code analysis is performed on static code, which is not being executed during test. While this has the benefit of full test coverage, it cannot easily provide a test verdict for complex problems and therefore will result in a lot of false positives. Fuzzing, on the other hand, needs to be executed against executable code. It can find vulnerabilities that are triggered by complex runtime behavior. Such vulnerabilities are not necessarily visible in the static code. Fuzzing can be performed against live systems. However, because fuzzing is an intrusive method of testing and will most probably crash the target system, it is recommended that you at least start testing in a separate environment, for example a virtualized environment or a test lab. Figure 2 illustrates the differences between the testing techniques by comparing the types of bugs they find.

1	2	3	4	code
				<pre>#include <stdio.h> int main() { char buffer[20]; printf ("Please enter your name."); scanf("%s", buffer); printf ("your name is %s \n", buf); }</pre>
				<pre>extern int name; void f(char *name); // declaration: no problem here // ... void f(char *arg) { // definition: no problem, arg doesn't hide name // use arg }</pre>
				<pre>pad = ciphertext.data[ciphertext.size - 1] + 1; /* pad */ length = ciphertext.size - hash_size - pad; if (pad > ciphertext.size - hash_size) for (i = 2; i < pad; i++) { if (ciphertext.data[ciphertext.size - i] != ciphertext.data[ciphertext.size - 1]) pad_failed = GNUTLS_E_DECRYPTION_FAILED; }</pre>

Figure 2: Bugs Discovered by Fuzzing and Static Code Analysis

- 1 - BUGS FOUND BY **STATIC ANALYSIS**
- 2 - BUGS FOUND BY **FUZZING**
- 3 - BUGS ACCESSIBLE BY **OPEN INTERFACE**
- 4 - BUGS RELATED TO **COMPLEX RUNTIME BEHAVIOR**

7 Vulnerability Management

Crisis Management is Expensive

In traditional vulnerability management, most of the costs arise from patch deployment. This not only holds for the vendor-side, but also for the user-side. Patch-deployment is always carried out as a crisis management process. Software developers race against the clock in an attempt to first find and verify the vulnerabilities, and then deploy workarounds or patches for their customers. Often the quality of these patches cannot be vouched for. For example, if there is a buffer overflow vulnerability, the programmers might just increase the size of a buffer without fixing the actual problem. Maintenance downtime during patch deployment is also expensive.

Being Prepared Pays off

In order to improve the security of the software they use, companies need to move away from reacting to somebody else's security discoveries to proactively finding vulnerabilities. In reactive security, patches are usually implemented in later stages making the whole process more costly. By detecting attack vectors and discovering zero-day vulnerabilities proactively you can stay ahead of the industry and proactively protect yourself against zero-day weaknesses. Fuzzing can also be used to test around patches to verify their quality. But one of the key benefits of proactive testing from a cost saving perspective is that you can do all the testing and patching at your own pace, and then push all the corrections and updates

to your services and systems in one big security push, thus reducing the amount of downtime needed.

Identifying Vulnerabilities

There are open source tools available for both fuzzing and static code analysis. However, in addition to their testing capabilities, their reporting capabilities are also limited compared to more advanced commercial solutions, which not only enable better test coverage and faster test runs, but also provide extensive test case documentation, which makes the repair process easier. For example, they commonly utilize Common Weakness Enumeration (CWE), which is an industry standard, to describe the vulnerabilities. This significantly facilitates the process of identifying and prioritizing the found vulnerabilities.

Fixing Zero-Day Vulnerabilities

If you find critical zero-day vulnerabilities in your own code, the most critical question is how widely employed and how public your product is. If you use open source projects, then the whole process is public, also to potential hackers, who will seize the opportunity to create exploits. After you release a patch, it is a race between your end users employing your update and the attackers seeking to exploit it. You might, for example, consider launching a pre-warning system, at least for your most critical customers, so they know to expect any major security updates. If you find issues in third party software, the important thing is to notify the vendor, so that they can fix the problem and to protect your own systems. In both cases, it is always advisable to contact your local CERT organization, which will assist you in the disclosure process.

8

Fuzzing and Static Code Analysis in the SDL and BSIMM Models

Security Perspective

While the SDL defines the product security process for software developers, the BSIMM model looks at security from a quality perspective. It depicts an organizational maturity model that helps companies benchmark their product security practices to state-of-the-art practices in the industry. The BSIMM model stresses the importance of static code analysis, because it ensures that secure coding policies are being followed. Protocol fuzzers, on the other hand, are essential, because they encapsulate an attacker's perspective. BSIMM is a product security checklist listing different activities that you might want to include in your own processes, but it does not enforce any activity over others.

Development Lifecycle Phases

One big difference between the BSIMM and the SDL model is that, unlike the SDL model, the BSIMM model does not attempt to place any of the security practices in a certain phases of the SDLC. In the SDL model, static code analysis is used in the Implementation phase and fuzzing in the Verification phase, as can be seen in Figure 3. However, both assessment methods can also be used in other lifecycle phases as well.

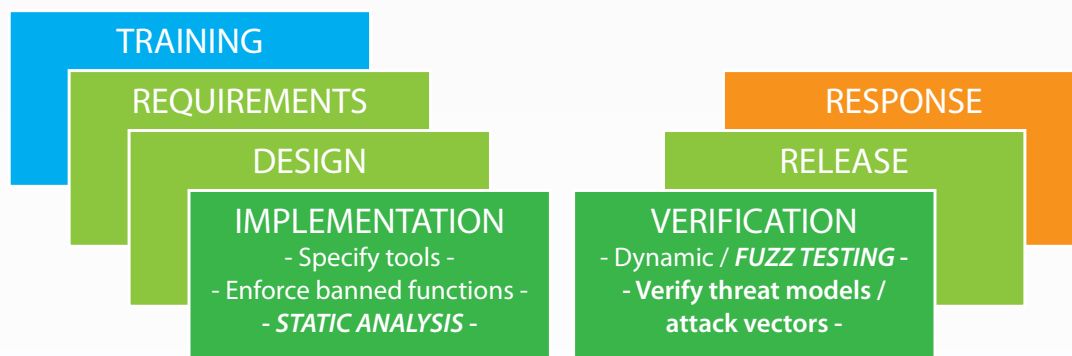


Figure 3: Static Analysis and Fuzzing in the SDL

For software testers the most important phase in the SDL is the verification phase, where fuzzing is used to test the robustness of the system before the release. However, fuzzing can be used throughout the development process from the moment the first software components are ready and even after the release. In fact, the earlier you start fuzzing the easier and cheaper it is to fix the found vulnerabilities. Similarly, the earlier you start using static code analysis the better the results you can achieve. Some advanced approaches even enable you to perform static code analysis directly after the code has been written in a development environment.

9 Combining Fuzzing and Static Analysis

Complementary Testing Methods

Static code analysis covers the entire source code, which makes it a very exhaustive form of testing. Especially, when testing larger scale systems, the amount of found bugs makes it difficult to determine which bugs are actually critical is difficult. Therefore, static code analysis can be a very slow and time consuming process. Fuzzing helps streamline the testing process by focusing the testing effort on the most critical interfaces. Fuzzing is a risk-based testing approach. It is a very representative method for testing software robustness, because it targets the problems attackers would also find. However, some bugs, like problems in error handling when the disc space is full, are triggered through an external interface, yet they are easier to discover with static code analysis, because testing them by fuzzing would require a vast amount of test cases in several different test configurations.

Use Cases for Fuzzing and Static Code Analysis

Fuzzing and static code analysis can be combined easily. When a new type of vulnerability is found by fuzzing, the fingerprint of the vulnerability can be taught to the static code analysis tools. This will help in potentially eliminating similar mistakes in other branches of the code that might not be exposed to attacks at the moment, but can become critical in future releases of the same software. Also, when a vendor releases a patch, you can use fuzzing to test around the patch to test its effectiveness. The benefits from using both fuzzing and static code analysis are derived from the differences in the testing technologies.

10 Conclusion

The purpose of proactive testing is to find vulnerabilities, before somebody else finds them and exploits them. By integrating fuzzing and static analysis 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.

Proactive testing is not just about protecting systems against attacks. Cleaning up your code also serves another purpose. Namely, it improves the quality of your software. As a white-box testing technique, static code analysis can be used to improve the quality of in-house software developments, but fuzzing can also be used 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.

Static code analysis and fuzzing are complementary in many aspects. While code analysis is static, fuzzing is dynamic. Static analysis is white-box testing, while fuzzing is mainly black-box testing. These two methods complement each other by finding vulnerabilities that could not have been found using just one method. Static analysis can easily find bugs, which would require vast amounts of feeds to uncover with fuzzing. Fuzzing, on the other hand, can find bugs that reveal themselves when the code is executed, for example vulnerabilities in dynamic libraries or bugs created by the code compiler or hardware setup. Overall, using multiple testing methods improves the test coverage as both the BSIMM and SDL models advocate.

Test Suites

Core Internet	Net Management	Routing
IPv4 (TCP, UDP, IPv4, ICMP, IGMP, ARP), IPv6 (TCP, UDP, IPv6, ICMPv6), IPsec, DNS-SEC, NTP (Client, Server), DHCP/BOOTP Client, DHCP/BOOTP Server, HTTP Server, HTTP Client, FTP Server, DHCPv6 Client, DHCPv6 Server, MIPv6 (Client, Server)	HTTP Server, HTTP Client, TLS/SSL Server, TLS/SSL Client, Telnet Server, SSH1 Server, SSH2 Server, SNMPv1/v2 Server, SNMPv3 Server, TFTP Server, UPnP Server, Syslog, SNMP TRAP	IS-IS, DVMRP, GRE, OSPFv2, OSPFv3, PIM-SM/DM, RSVP, VRRP, BGP4, RIP, RIPng, MPLS/LDP, HSRP, NHRP

Remote Access	VPN	VoIP/IMS
EAPOL Server, PPPoE, Diameter Server, Diameter Client, LDAPv3 Server, TACACS+ Server, TACACS+ NAS, RADIUS (Server, Client), Kerberos Server	IPSec, SSH1 Server, SSH2 Server, TLS/SSL Server, TLS/SSL Client, ISAKMP/IKEv1 (Client, Server), IKEv2, OCSP (Client, Server), L2TPv2, L2TPv3, x.509	SCTP, H.248, H.323, RTSP (Client,Server), TLS/SSL Server, TLS/SSL Client, SIP UAS, SIP UAC, SigComp, RTP/RTCP/SRTP, MGCP, UPnP Server, SMPP, x.509, BICC, STUN, TURN

3G / 4G-LTE	Digital Media	Email
SCTP, GRE, IPsec, Diameter (Server, Client), LDAP Server, TLS /SSL (Server, Client), SIP UAS, SIP UAC, GTPv0, GTPv1, GTPv2, RADIUS (Server, Client), PMIP	AIFF, AU, AMR, IMY, MP3, VOC, WAV, BMP, GIF, JPEG, MBM, PCX, PNG, PIX, PNM, RAS, TIFF, WBMP, XBM, XPM, WMF, AVI, Quicktime, MPG1, MPG2, MPEG4, ZIP, CAB, JAR, LHA, GZIP, vCalendar, VCard	POP3 Client, POP3 Server, IMAP4 Client, IMAP4 Server, SMTP Client, SMTP Server, MIME

File Systems/Storage	WLAN	Link Management
CIFS/SMB Server, iSCSI Server, SunRPC Server, NFS Server, SMBv2, FCoE, FIP, PFC	802.11 Server, 802.11 Client, WPA Server, WPA Client	LACP, STP, MSTP, RSTP, ESTP

Bluetooth	IPTV	PDA/ Smartphone
L2CAP, SDP, RFCOMM, OBEX, OPP, FTP, IrMC Sync, BIP, BPP, BNEP, HFP, HSP, DUN, PBAP, FAX, AVRCP, A2DP, HCRP, HID, SAP, HFP Client, HSP Client, BPP, MDP/HDP, 2.1 compliant	MPEG4, MPEG2, IPsec, TLS/SSL, RTP/RTCP, RTSP, HTTP, FTP, TFTP, IPv4, PIM-SM/DM, RSVP, IGMP, CWMP(TR-69), MPEG2-TS, SIP-UAS, SIP-UAC	IPv4, DHCP/BOOTP, HTTP, TLS/SSL, UPnP, SIP, Audio, Images, Video, Bluetooth, 802.11

Industrial Automation	Archives	Metro Ethernet
(SCADA/DCS) Modbus, IPv4 (TCP, UDP, IPv4, ICMP, IGMP, ARP)	GAB, GZIP, JAR, LHA, ZIP	BFD, CFM, E-LMI, Ethernet, GARP, LLDP, OAM, PBT/PBB-TE, STP/RSTP/MSTP/ESTP, PTP, SyncEthernet

General Fuzzers
XML (File, SOAP), Traffic Capture Fuzzer, Generic File Format Fuzzer

Protocols Supported

802.11	IPv4	RSTP
ARP	IPv6	RSVP
BGP4	IS-IS	RTCP
BFD	ISAKMP/IKE	RTP
BICC	iSCSI	RTSP
BOOTP	Kerberos	SCTP
BT	L2TPv2	SigComp
CFM	L2TPv3	SIP
CIFS/SMB	LACP	SIT TT
CWMP (TR-69)	LDAPv3	SMBv2
DHCP	LLDP	SMPP
DHCPv6	LPD	SMTP
Diameter	MGCP	SNMPv1
DNS-SEC	MIPv6	SNMPv2c
DVMRP	MIME	SNMPv3
E-LMI	Modbus	SRTP
EAP	MPEG2-TS	SSH1
ESTP	MPLS/LDP	SSH2
Ethernet	MSTP	STP
Finger	NFS	STUN
GARP	NetBIOS	SunRPC
FCoE	NHRP	SyncEthernet
FIP	NTP	Syslog
FTP	OAM	TACACS+
GRE	OCSP	TCP
GTPv0	OSPFv2	Telnet
GTPv1	OSPFv3	TFTP
GTPv2	PBT/PBB-TE	TLS/SSL
H.248	PFC	TURN
H.323	PIM-DM/SM	UDP
HSRP	PMIP	UPnP
HTTP	POP3	VLACP
ICMP	PPPoE	WPA1 WPA2
ICMPv6	PTP	VRRP
IGMP	RADIUS	X.509
IKEv2	RIP	XML
IMAP4	RIPng	
IPsec	Rlogin	

Technical Requirements

Supported Operating Systems
Windows 7, Windows XP SP3 and Linux CentOS

Minimum System Requirements
1 GHz processor, 1 GB of free disk space, 1024x768 graphics resolution, 1 GB of RAM, CD-ROM or DVD drive, Network card (NIC)

Oracle Java™ 2 Runtime Environment
Standard Edition 6 (1.6) 32-bit

USB port required for Bluetooth and WLAN tools

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

Tutkijantie 4E
FIN-90590 OULU
FINLAND
+358 424 7431

10670 North Tantau Avenue
Cupertino, CA 95014
UNITED STATES
+1 408 252 4000

25/F., Queen's Road Centre
152 Queen's Road Central
HONG KONG
+852 3426 22900