MDISS Technical White Paper Series

Fuzz Testing: Improving Medical Device Quality and Safety

Contents

1 Introduction

This document contains a framework for improving the safety and reliability of software and firmware in medical devices by using *fuzzing*. Fuzzing is a testing technique for locating unknown vulnerabilities and other defects by sending malformed and unexpected inputs to software. After a brief introduction to fuzz testing, and an examination of how different stakeholders would use fuzzing, and how fuzzing can be used to harden medical devices and networks against attack, the framework is presented. The framework is designed to be a fully documented process with repeatable results, suitable for hardening medical devices, software and networks against attack, improving their quality, assessing risk, measuring progress, and allowing meaningful comparisons between similar systems.

Software and firmware are woven into the fabric of society. While technological advances can be exhilarating, corresponding risks have also emerged. When software or firmware does not work as intended, or when attackers use software and firmware for their own purposes, the consequences can be severe.

In this document, the term *software* means any software or firmware, in essence any executable computer instructions, no matter in what type of device or system they reside.

Nowhere is this risk more immediate than in the medical community. While the landscape of medical devices is very diverse, the software that runs each of these devices impacts quality of care and patient safety. Software quality and the process of hardening medical device systems is important for all devices, be they implantable pacemakers, surgical robots, large machines delivering precise doses of life-saving radiation, or the electronic health records systems that must safeguard protected health information (PHI) and the integrity of data that is used for clinical decision making.

The increasing complexity of software mandates vigilance to ensure that software works as intended. Unfortunately, software frequently fails in the face of unexpected or malformed inputs, also called *fuzz*. Fuzz can happen when software encounters real world conditions, such as interacting with humans or machines that don't behave as expected. Fuzz can also happen deliberately if an attacker wishes to gain control of a system or disrupt the normal operation of a piece of equipment.

The Institute of Electrical and Electronics Engineers (IEEE) defines *robustness* as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.[1] Having a properly functioning system, despite the unpredictable, is essential in the medical systems world in order to keep patients alive and information confidential.

Software bugs (bugs) often create vulnerabilities because they cause software to behave differently than intended. The software might crash, making it unavailable, or consume all available resources, or cause other unpredictable consequences. In the worst case, an attacker might be able to trigger the bug in a special way such that he or she can run his or her own instructions. Some bugs are exposed and fixed during the testing phase of a software development process. The bugs that slip past the testing

[1] "Standard Glossary of Software Engineering Terminology (ANSI)". The Institute of Electrical and Electronics Engineers Inc. 1991.

phase without being found and fixed are *unknown vulnerabilities* and can be triggered, sometimes with catastrophic results, after the product release.

Fuzz testing, or *fuzzing*, is the process of sending intentionally malformed inputs to software for the purpose of locating vulnerabilities. When failures are found, they can be fixed, which makes the software more robust and more secure.

2 Who Fuzzes?

Fuzzing is appropriate for medical device manufacturers, computing and network equipment manufacturers, healthcare delivery organizations (HDOs), researchers, and the organizations that live in the medical industry supply chain. In general terms, these organizations are *builders*, *buyers*, or a mix of both roles.

A *builder* is an organization that creates software, such as an insulin pump manufacturer or an electronic record system manufacturer. Builders can improve the robustness and security of their products by incorporating fuzzing into their software development life cycles. Fuzzing is a testing technique and complements other types of testing that are already part of the Quality Assurance (QA) portion of the software life cycle. Fuzzing locates vulnerabilities and provides the software development team an opportunity to fix these vulnerabilities before the product is released.

In an ideal world, all medical device manufacturers would perform fuzzing as part of their software development life cycle, resulting in products that are more robust in the face of unexpected real world conditions, and more secure in the face of attackers. In reality, the health care community is just starting to realize the importance of security in networked devices, which means that medical devices are likely to have been created in the absence of any fuzzing.

Keep in mind that devices that are interconnected in any way, or take any type of input, are good candidates for fuzzing. Devices might be connected with physical wires (Ethernet and serial connections) or wireless technology (Bluetooth, ZigBee, and 802.11). Even devices with proprietary wired or wireless technologies can and should be fuzzed.

A *buyer* is an organization that purchases equipment from builders. Buyers incorporate fuzzing into their evaluation and verification processes. In a medical setting, the buyer is an HDO. A hospital, for example, purchases medical devices, equipment, and computer systems. A hospital making a medical device purchase should confirm that fuzz testing has been performed as part of the quality assurance process for the product.

The availability of fuzz testing during the product evaluation phase of procurement is especially important for medical devices. The remediation of discovered defects or vulnerabilities may require a significant amount of time for the manufacturer to address the vulnerability and then to pursue recertification of the updated software by the FDA.

In addition, a hospital with an existing network of computers and systems could use fuzzing to assess their risk of attack. The hospital should discuss with a security expert how to use fuzzing to evaluate the software in use in their environment. This evaluation must be done in a testing laboratory and never in a production environment.

A better alternative for HDOs, instead of performing fuzz testing themselves, is to compel their manufacturers to perform fuzz testing during product development. The HDO can provide specific requirements on the type of fuzzing that must be performed and require documentation of the results.

Fuzzing must *never* be performed on systems or equipment in use for patient care, or on medical devices or related systems that will subsequently be used for patient care.

In the medical industry, a supply chain often exists between builders and buyers. For example, an integrator might acquire components from other vendors and assemble them into a product, which is ultimately purchased and used by an HDO. An integrator could use fuzzing (again, as part of the software development life cycle) both to verify the software in acquired components as well as to improve the robustness and security of the assembled product.

In addition to legitimate users, individuals or organizations wishing to steal information or disrupt healthcare delivery also use fuzzing.

2.1 Success Story: Intuitive Surgical

Intuitive Surgical makes a minimally invasive surgical robot, the *da Vinci* Surgical System:

http://www.intuitivesurgical.com/products/davinci_surgical_system/

da Vinci includes some innovations that leverage the power of the Internet. In particular, the system can "phone home" to Intuitive Surgical, making it easy for support staff to answer questions or diagnose problems with a particular installation. The *da Vinci* system also allows remote mentoring or proctoring, where a surgeon in one location can observe and assist a surgeon in another location. Both the servicing and mentoring features use standard Internet protocols and therefore are good candidates for fuzz testing.

Brian Miller, Senior Director of Advanced Development at Intuitive Surgical, described one example where fuzz testing helped pinpoint a problem. During development of the remote servicing capability, where *da Vinci* systems periodically communicate with Intuitive Surgical, some systems stopped communicating with the server, making the remote servicing capability inaccessible.

Fuzz testing revealed that the system did not recognize a certain kind of protocol message, in which case the software was designed to stop communicating with the server. "We were able to troubleshoot the software and figure out why it wasn't able to handle the response and ultimately fix it," explains Miller. "So with that I was sold that fuzzing was something we needed to integrate into our verification test process."

2.2 Success Story: Verizon

Fuzzing is in widespread use in the telecommunications industry. Large network providers use equipment from many different vendors to build sophisticated telecommunication networks. Service availability is paramount.

The story of Verizon helps illustrate how a buyer organization introduced fuzzing into a specific industry and dispersed the technique to its vendors. It is a model that could be applicable to HDOs in the health care industry.

Verizon uses fuzz testing as a way of selecting equipment vendors. After purchases are made, Verizon uses fuzz testing to probe for unknown vulnerabilities. Found vulnerabilities are communicated to network equipment manufacturers so their products can be made more robust and secure, and thus work better and more reliably.

To improve the efficiency of fixing vulnerabilities, Verizon now requires that its vendors do their own fuzz testing before submitting equipment into the Verizon testing lab. This is a win for everyone involved. Verizon's vendors save money because they are fixing bugs much earlier in their development cycle. In addition, their products are more robust and more secure. Verizon benefits because they are now able to purchase more robust and more secure products, which ultimately results in better network reliability and lower customer service costs.

3 Modus Operandi of the Black Hat

How do malicious attackers, or *black hats,* take control of your network or your devices? It starts with software bugs. A black hat who wishes to infiltrate your network will follow these steps:

1. Search for appropriate target software.
2. Fuzz the target software by sending malformed and unexpected inputs. When the target software fails, the black hat has found a vulnerability, or bug. If no one else knows about the vulnerability, it is an *unknown* or *zero-day* vulnerability.
3. For each vulnerability found, the black hat will perform additional research to see if the target software can be made to fail in a specific way, such that the black hat can gain control. When the black hat succeeds, he or she has created a *zero-day exploit*.
4. The black hat chooses a method to deliver the zero-day exploit to software inside your network. One common method is *spear phishing*, in which the attacker forges an email to someone in your organization and convinces them to open an attached file or click on a link that infects the victim's machine.
5. The black hat now has access to one system in your network and can now perform surveillance, attack other systems, or cause other kinds of trouble.

The operations of black hats are not theoretical. Attackers are taking advantage of insecure software across all industries for various purposes. Attackers range from individuals looking for mischief, through criminal organizations looking for profit, to state-sponsored groups intending to disrupt critical infrastructure or cause individual harm through medical means. Here is one example of a Bluetooth-based attack on a small device:

http://www.youtube.com/watch?v=8d7pC9WmQ-U

4 Does This Software Make My Attack Surface Look Fat?

Every place you have software is a possible entry point for black hats. Software is designed to take input and make something happen. Black hats attempt to find ways to misuse software so that, instead of doing what it is supposed to do, it carries out the attacker's instructions.

Many applications take input over standard network protocols, and many applications take input in the form of files with a standard format. Protocol specifications describe the structure and sequence of messages exchanged between two communicating pieces of software. Any software that understands a protocol or format is subject to fuzzing, where deliberately malformed messages or files are delivered to the software to probe for vulnerabilities. Software that is not specifically written to be robust will often fail when presented with protocol messages that do not conform to the specification.

Your *attack surface* is the sum of all the places in your network where any kind of software takes any kind of input.

In a typical hospital, for example, software runs on desktop and laptop computers. Many other parts of the network are also attack vectors. Printers are an increasingly popular target, being full-fledged members of the network but often running outdated, buggy software. In addition, printers offer a variety of input options and thus expose a broad attack surface in terms of network protocols as well as file formats. WiFi access points also expose a variety of protocols. Hospitals might also carry telephone conversations over the network, so telephone exchanges and telephones themselves can also be attack vectors.

Medical devices such as X-ray machines and MRI equipment communicate with other smart medical equipment, and the protocols and file formats used between different machines are excellent candidates for fuzzing. Any machine that deals with DICOM, for example, should be fuzzed.

Back office systems such as accounting and personnel systems are also targets for attack, especially when their functionality is exposed to the Internet through a Web server. Web servers themselves typically expose a variety of protocols, each of which is a target for fuzzing.

Mobile devices like smart phones and tablets are quickly becoming popular attack targets. Many organizations allow mobile devices access to internal networks and applications. If the mobile device can be compromised by a black hat, then he or she can use it as a springboard to attack the rest of your network.

Hospitals have systems that track patient records and systems for billing and insurance. Protecting patient confidentiality and preventing data leakage are primary concerns. All these systems take input in one form or another and can be fuzzed to locate vulnerabilities.

Finally, smaller devices like infusion pumps and glucose meters contain software and may support handfuls of Bluetooth, WiFi, and other protocols, all of which can be fuzzed.

5 All Software Has Bugs

All software has bugs—it's inescapable. The best developers in the world will nevertheless write software with bugs. Modern software is tens, often hundreds of thousands of lines of code, with more complex software systems reaching into the millions.

Black hats search for bugs by fuzzing, sending malformed and unexpected inputs to software. Part of the reason this is so effective is that fuzzing is very different from traditional software testing.

The usual software development life cycle has developers writing software and testers testing it. If the testers find a bug, they make a record of it and the developers do more work to fix the recorded bugs. But what exactly are the testers doing?

Traditional software testing is *positive* testing. If you give the software X input, does it produce Y output? Traditional software testing verifies that the software behaves in the way you want when you give it valid input.

Fuzzing, by contrast, is *negative* testing. If you supply malformed input to the software, does it crash? Does it slow down? Does it start behaving oddly?

If software testers only perform positive testing, the software has vulnerabilities that will only be exposed during fuzzing, which means black hats are easily able to locate and exploit zero-day vulnerabilities in the software.

In addition to security, vulnerabilities are exposed when unexpected or malformed inputs are delivered to software through real-world conditions. In this case, the bugs are discussed in terms of *quality* or *robustness* rather than security. In the medical community, the quality and robustness of software has a direct effect on healthcare quality.

6 You Can Build a Wall, But It Won't Work

Throughout history, humans have defended themselves by building a perimeter to keep out attackers. Think of walled medieval European cities, or the Great Wall of China, or a simple mound of dirt with pointy sticks on top. The same idea has been applied to computer networking. Companies deploy firewalls, antivirus software, and Intrusion Detection Systems (IDS) around the perimeter of their network.

While perimeter defenses offer legitimate protection against the threats they are designed to prevent, they are ineffective at protecting against unknown vulnerabilities for two reasons.

First, perimeter defenses are all based on blocking *known* attacks. Antivirus scanners and IDS systems mostly work by telling you about styles of attack that everyone already recognizes. The real threat is *unknown vulnerabilities*, or *zero-day* attacks. These are freshly made attacks—attacks that no one has seen before. Perimeter defenses offer little protection against such attacks. Iran could have used antivirus scanners on all their computers at Natanz every fifteen minutes, and Stuxnet would still have come barreling in, because it was based on zero-day vulnerabilities.

Second, in the world of computer networking, the very concept of a perimeter is chimerical. It's a line drawn on network diagrams and presentation slides; it represents a physical concept that has little meaning in computer networks. Individual computers that appear to be inside the perimeter are capable of making direct connections with computers worldwide, essentially hopping right over the perimeter and interacting directly with systems that might or might not be trustworthy. Users can easily click links or navigate to web pages that install malware on their systems, inside the perimeter.

7 Fuzzing for Defense

One of the best defenses against black hats is to do your own fuzzing to locate and remediate zero-day vulnerabilities. If you fuzz your own software and fix the vulnerabilities you find, then black hats will have a much more difficult time locating vulnerabilities. Fuzz testing software is, in some sense, analogous to clinical testing on a new drug.

Note that the quality of your fuzz tool needs to be at least equivalent to the quality of the attacker's fuzz tool. Fuzzing comes in several different varieties, and the capabilities of fuzz tools vary widely. If you test your software with two million fuzzed inputs, but they are not very effective, an attacker might still discover vulnerabilities with only a few thousand well-crafted fuzzed inputs. Each malformed or unexpected input is a *test case*.

Also note that different fuzzing techniques find different vulnerabilities. *Template fuzzing* works by replaying a protocol conversation many times, each time introducing anomalies into the recorded template. *Generational fuzzing* uses knowledge of the protocol, based on its specifications, to generate test cases for every field of every message type. Therefore, the best defense is to exercise your software with multiple fuzzing techniques in order to find and fix the maximum number of vulnerabilities.

Ideally, you will fuzz the entire attack surface of your software or organization. Network protocols always work in layers, with each layer being represented by a different body of code, so the best way to find the most bugs is to fuzz every layer. For example, a typical web server understands HTTP, but HTTP runs over TLS, which runs over TCP, which runs over IP. Each protocol layer has code which might contain vulnerabilities. An insulin pump might have an application-specific protocol that runs on top of Bluetooth Medical Device Profile (MDP). This would mean that testing your entire attack surface would entail fuzzing L2CAP, MDP and the SDP layer.

Remember, nothing in security is absolute. You will never find and fix all the bugs in a piece of software. All you are trying to do is make it cost more to break in to your network than the value of potential gains. At the very least, you will slow down or deflect an attack, giving you a better chance of detecting it and combatting black hats directly.

7.1 Return on Investment: Financial

Fuzzing makes excellent financial sense. For organizations that create software, like device manufacturers, it costs far less to fix bugs during the software development cycle than to fix bugs after products have been released. After products are sold and deployed, bugs found "in the wild" incur significant costs to the manufacturer in terms of support, patches, and customer advisories. Finding and fixing more bugs during the development cycle results in measurable cost savings.

7.2 Return on Investment: Patient Safety and Reputation

Other benefits of fuzzing are more important, but harder to quantify. If your product fails because of a vulnerability that you did not fix, patients could die or suffer significant morbidity. Likewise, your company reputation might be damaged beyond repair, not to mention the possibility of class action lawsuits and allegations of gross negligence. Device manufacturers can protect patients, their customers, their shareholders, and themselves by integrating fuzzing into their software development life cycles to create software that is more robust and more secure.

8 Deployment Options

Organizations wishing to get started with fuzzing have four options. Remember, a fuzzer is just a software test tool, so the choices are essentially the same as for any other test tool.

1. **Build a fuzzer**. This gives you complete control and a high level of confidentiality, but is difficult and expensive. Fuzzing expertise is hard to find and retain, and the long-term costs of building and maintaining your own fuzzer are formidable. In addition, you'll need to figure out how to verify that your homegrown solution is really effective in testing your software.

2. **Pay for fuzzing**. You might decide it makes sense to pay a third party to fuzz your software. This frees you from needing any in-house fuzzing expertise, but it can be clumsy. In the early stages of software development, testing iteration is crucial, and iterating with a third-party testing service limits your agility. Also, it can be difficult to evaluate the effectiveness and expertise of the service you choose.

3. **Pay for fuzzing in the cloud**. This is a variation on #2, where you upload your software to a fuzzing service online, send money, and receive results. This type of service is easier to use, more nimble than a traditional fuzzing service, and enables you to explore the benefits of fuzzing with minimal investment. However, it has the same disadvantages as #2, and simply will not work for certain types of targets. A physical device like a glucose meter, for example, cannot be uploaded to an online fuzzing service, although you might be able to fuzz some of its software components separately.

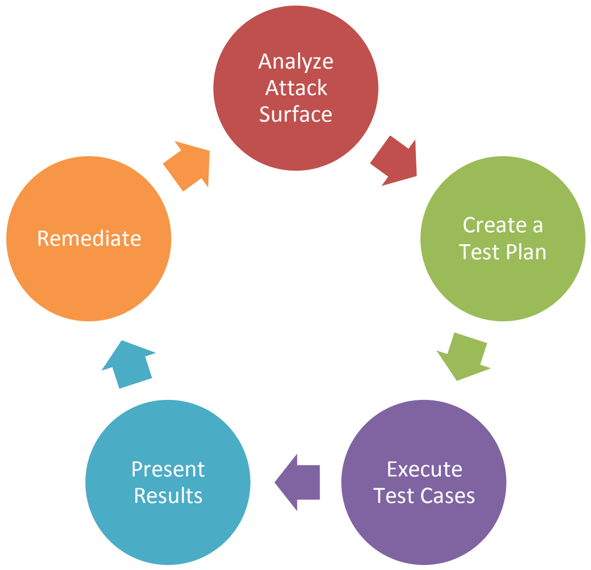
4. **Buy a fuzzer**. When you own a fuzzer, all testing happens in-house. This allows you to integrate fuzzing into a software development or verification life cycle without having to develop or acquire extensive fuzzing expertise. Having the testing in-house for software development means you'll have a tight feedback loop and can quickly find and fix vulnerabilities.

A good fuzzing tool provides the following features:

- **Reproducible results**. If the tool causes a failure, you should be able to easily deliver the same test cases to the target to reproduce the failure.
- **Good reporting**. Communication is crucial to getting vulnerabilities fixed. A good tool provides informative, easy-to-read reports that transfer information efficiently between test teams and development teams.
- **Effective fuzzing**. Ten thousand carefully crafted generational test cases will probably be more effective than one million random test cases. The most effective fuzzing is usually generational. Also, don't forget that experience counts. A more mature tool is likely to be more clever about how it generates test cases than a new tool.
- **Ease of use**. When you purchase a fuzzer, part of what you are paying for is the ability to have minimal fuzzing expertise in your organization. The fuzzer should make it easy for people without much fuzzing knowledge to effectively locate vulnerabilities in software targets.

9 Fuzz Framework

This section presents a methodology for performing fuzzing to proactively secure and harden medical devices and software. Using this methodology and creating reports in a consistent format will allow you to assess your risk, measure progress over time, and perform meaningful, apples-to-apples comparisons of different products.



The framework consists of five steps:

1. Analyze Attack Surface
2. Create a Test Plan
3. Execute Test Cases
4. Present Results
5. Remediate

Each step is presented in detail in the coming sections.

9.1 Analyze Attack Surface

The first step is essentially reconnaissance and research. The *attack surface* of a target device or system is all the places where the target takes input. As you perform analysis of the attack surface, you will build a list of its *attack vectors*, which are all the ways the target accepts input from other computers or humans.

For targets that participate in IP networking, part of the attack surface analysis simply involves determining which ports are listening for incoming connections and what kinds of outgoing connections the target initiates.

A typical Web server, for example, listens for incoming HTTP protocol requests. Because network protocols work in layers, support for HTTP implies support for TCP and IP, and frequently TLS. Each protocol is represented by its own software, with its own bugs, which means that each protocol is a separate means of attack. Likewise, on top of HTTP is a web application, which might accept or deliver DICOM images, or use JSON to communicate from a browser to a patient information system.

By contrast, a glucose meter might accept incoming Bluetooth connections, which implies that each of its supported Bluetooth protocols are attack vectors.

Documentation of the attack surface should be included in the test plan produced in the next step.

9.2 Create a Test Plan

Documentation is important in fuzzing. Reliable documentation ensures that fuzzing is repeatable and reliable.

After analyzing the attack surface, you will need to make some decisions before you are able to create the test plan. First, you must decide what portion of the attack surface you wish to test. In an ideal world, you would fuzz every attack vector. Regrettably, the constraints of time and space sometimes dictate that you can only fuzz a subset of the full attack surface of your target. If this is the case, you need to sort your list of attack vectors by priority and, based on your available time and money, fuzz the most important attack vectors in your list.

Related to this decision about attack vectors is the availability of fuzz tools. Ideally, you have multiple fuzz tools, with multiple fuzzing techniques, available for each attack vector, and the time to run them all. In reality, you might have limited budget for acquiring tools and the ability to only run one tool for each of your attack vectors.

If you are able to run only one tool per attack vector, use a generational fuzzer.

Some generational fuzz test suites contain hundreds of thousands, even millions, of test cases. If you do not have time to run every test case, a reasonable baseline is to let the suite run for eight hours. Make sure to enable your fuzz tool's random execution mode to get a good distribution of test cases, which will result in reasonable test coverage in the target code.

The time required to run fuzz testing for a specific protocol depends on the protocol itself, the speed of the target, the speed of the test tool, and the speed of the network connection. Predictions for test times should be based on your previous experience with fuzzing.

A *test plan* is a document that describes what you are going to fuzz and how you are going to do it. It should include the following information:

- **Target device**. At a minimum, include the name of the target, its version, and any configuration that needs to be done to prepare the target for testing.
- **Attack vectors**. This is a list of the attack vectors that you will fuzz. For each attack vector, list the following:
  - **Tools**. Show the tools (including version numbers) that you will use to perform the fuzzing.
  - **Extent**. Detail how much fuzzing you will do. For generational fuzzers, indicate if you will limit the test run by time, for example by running for eight hours only. Alternately, indicate the number of test cases you will deliver to the target.

As you work through the the fuzzing framework, be sure to keep your test plan up-to-date. You might make adjustments as you go through the nuts and bolts of execution and analyze your results. Keep the test plan relevant and useful by maintaining it as an accurate representation of your work.

9.3 Execute Test Cases

Once the test plan is done, it’s time to execute! The exact details vary from tool to tool, but some general considerations apply to fuzz testing.

9.3.1 Testability

When performing fuzz testing, it is important to make sure the device is made testable. This means that typical countermeasures, such as firewalls or connection throttling, that prevent testing or at least slow down the tests should be disabled. The idea of fuzz testing is not to test the effectiveness of countermeasures, but to deliver the test cases to the system under test as efficiently as possible. This may also include enabling additional logging to be more easily able to determine if errors have occurred. The benefits of making the device testable usually greatly exceed those of not doing so in terms of the ability to reliably detect faults and execute tests faster. Thus, taking these additional steps is recommended.

Also note that testing on networked components might have unintended consequences. Delivering test cases to a target through intermediate network components might expose vulnerabilities in the intermediate components, causing crashes or other failures. Similarly, feeding test cases to a target system might cause failures on downstream systems that take input from the target. You might, for example, send bad inputs to a medical record database without incident, while a reporting system that pulls data from that database might experience a failure at some later time.

9.3.2 Instrumentation

In robustness/fuzz testing lingo, *instrumentation* refers to the process of monitoring the health of the target. In essence, instrumentation is used to determine whether the target has experienced an error, or whether it is still able to process incoming data robustly and securely.

A major challenge when instrumenting targets is that while a set of typical failure modes exist, exposing a system to fuzz tests can lead to unexpected and unforeseen failures, including fault propagation to other systems not specifically targeted. Furthermore, depending on the implementation language of the tested software, failure modes can vary greatly. While programs written in languages like C or C++ are more prone to segmentation faults or similar failures, programs written in languages like Java are likely to behave completely differently under stress.

In other forms of testing, such as conformance and load testing, pass or fail verdicts are determined against a set of expected behavior. This behavior is typically defined by means of target output, so there is always a yes-or-no answer. For example, the wrong response to a request is clearly a failure. In fuzz testing this type of approach is usually not applicable, and the pass or fail verdict is ultimately determined from how the target behaves. This means, in practice, that the only way to reliably assign a pass or fail verdict is to be able to monitor how the tested software behaves and if it shows vulnerable behavior. A process crash in the target software is almost always considered a failure, as is a kernel panic which causes a target to restart. Other modes of failure are harder to detect.

Good fuzz tools provide facilities for instrumenting and monitoring a target in multiple ways. The typical facilities that should be considered for instrumentation are:

1. Valid case instrumentation (if available)
2. Manual observation of console output, logs, front panel display, indicator lights, and any other visible indications of functionality
3. Automatic observation of the target via external instrumentation or scripts
4. Instrumentation via the Simple Network Management Protocol (SNMP)

We recommend that all tests should be executed with valid case instrumentation enabled, when it is available for particular test suite.

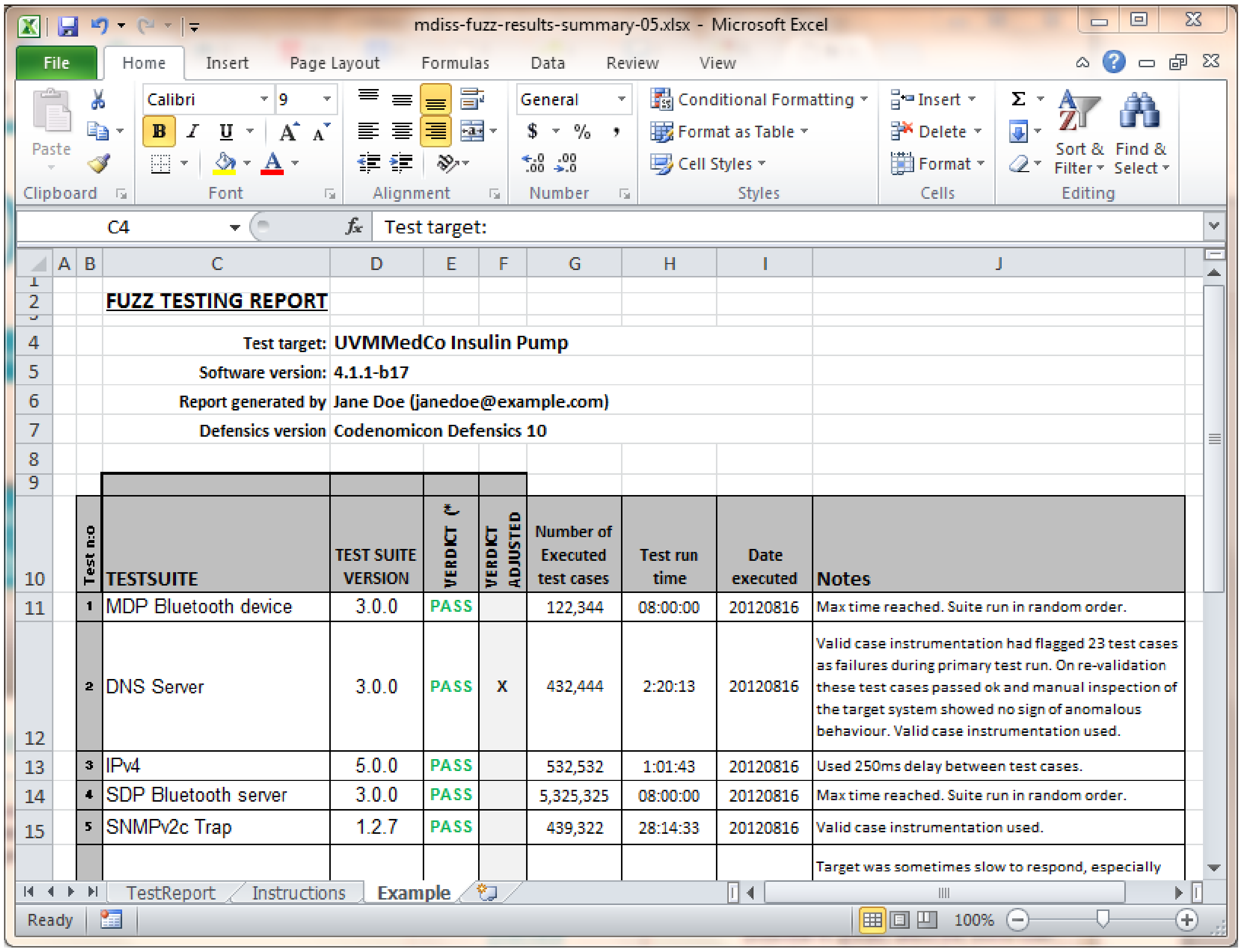
9.4 Present Results

In order to support comparing, tracing and archiving test results, this section contains guidelines on fuzz testing reporting format and content. Test results consist of two major parts:

- High level summary of the executed test suites
- Detailed test results and logs produced by the fuzz tool

This section describes the recommended format for the high level summary. The detailed test results are the output and logs of the fuzzer you are using and will vary from tool to tool.

The high level summary condenses the executed test runs, the primary results, and test run parameters into a single, easy-to-fill and easy-to-read document. Reading the summary provides a good understanding of the tests performed and the maturity level of the target systems. If errors have been noted, or other anomalies have been noted during the test runs, recording these to the **Notes** field in the summary is crucial.



mdiss-fuzz-results-summary-05.xlsx - Microsoft Excel

C4 Test target:

FUZZ TESTING REPORT

Test target: UVMMedCo Insulin Pump
Software version: 4.1.1-b17
Report generated by Jane Doe (janedoe@example.com)
Defensics version Codenomicon Defensics 10

Test n:o	TESTSUITE	TEST SUITE VERSION	VERDICT	VERDICT ADJUSTED	Number of Executed test cases	Test run time	Date executed	Notes
1	MDP Bluetooth device	3.0.0	PASS		122,344	08:00:00	20120816	Max time reached. Suite run in random order.
2	DNS Server	3.0.0	PASS	X	432,444	2:20:13	20120816	Valid case instrumentation had flagged 23 test cases as failures during primary test run. On re-validation these test cases passed ok and manual inspection of the target system showed no sign of anomalous behaviour. Valid case instrumentation used.
3	IPv4	5.0.0	PASS		532,532	1:01:43	20120816	Used 250ms delay between test cases.
4	SDP Bluetooth server	3.0.0	PASS		5,325,325	08:00:00	20120816	Max time reached. Suite run in random order.
5	SNMPv2c Trap	1.2.7	PASS		439,322	28:14:33	20120816	Valid case instrumentation used.
								Target was sometimes slow to respond, especially

Image 2.1 – Example results

The fields should be interpreted (and filled) as follows:

- **SUT** - Describe the name of the target, or system under test (SUT). It usually should contain the name of the system you are testing.
- **Software version** - Uniquely identifies the software or software packages running on the system under test.
- **Report generated by** - Optional element that should contain name and contact details of the person who generated the report.
- **Test suite version** - Version of the tool used for the fuzz tests.
- **Test no** - Fixed field that identifies the test run number on the Excel sheet.
- **TEST SUITE** - Test suite identifier that uniquely identifies the test suite used.
- **TEST SUITE VERSION** - Version of the test suite used.

- **VERDICT** - one of PASS or FAIL.
- **VERDICT ADJUSTED** - Check (X) this column if you manually altered the verdict from FAIL to PASS. This should only be done if persistent false positives are noted during test run which after manual verification cannot be identified as real faults. If this field is checked, then 'Notes' column MUST contain explanation for the adjustment.
- **Number of executed test cases** - Total number of fuzz tests executed for the tested interface during the test run. Filling this field is optional.
- **Test run time** - Total time spent running the tests in 'Number of Executed test cases' column. Filling this field is optional.
- **Date executed** - Date when tests were executed. Use Test run end date for this.
- **Notes** - Use this field to describe and summarize the test run. Especially if any anomalous behavior was noted, this is the field where you should be as verbose as possible to describe what happened. Reviewing notes fields should give the reader a good grasp of what transpired during the test run. Use this field for additional information such as the actual interface, blade, etc. that was targeted. Also, additional test scenario information should be included here. If **Verdict adjusted** has been checked, then this field should ALWAYS contain an explanation for verdict adjustment.

9.5 Remediate

Each of the failed test cases found during testing represents a vulnerability. The final step in the fuzzing framework is to communicate results to the team responsible for the target software so they can fix the bugs found.

If you develop your own software, you can integrate fuzzing into the development life cycle. After bugs are fixed, you can fuzz again for regression testing.

Likewise, if you're purchasing software from other vendors, you'll want to notify those vendors of any vulnerabilities you locate during fuzzing.

10 Conclusions

This document presents a framework for managing unknown vulnerabilities, based on fuzzing for improving the security and robustness of medical devices. Using fuzzing as part of a software development life cycle is a highly effective method of locating zero-day vulnerabilities, which can then be fixed. Using fuzzing as part of a verification or selection process gives you a clear picture of the robustness and security of medical devices and information systems.

Fuzzing is an excellent way to mitigate unknown threats in medical device networks and is a critical contribution to creating robust and secure medical devices and other technology products involved with patient care. Device manufacturers adopting this framework will be able to use increased safety to differentiate their products and reduce costs in terms of development and recalls. HDOs can also incorporate this framework to validate device security in their own environments. This framework provides a common foundation for them to communicate change requests back to the manufacturer.

The framework provides a solid methodology for fuzzing. The framework ensures that you have meaningful, repeatable results and are able to measure progress over time or compare, apples-to-apples, the performance of similar products. Future MDISS work will concentrate on advancing the framework and defining maturity levels for medical devices and other software.

Defend. Then deploy.

11 Contact Information

For more information, please contact:

Dr. Dale Nordenberg, MDISS

dalenordenberg@mdiss.org

Jonathan Knudsen, Codenomicon

jonathan@codenomicon.com

11.1 About MDISS

MDISS protects public health and well-being by advancing computer risk management practices to ensure wide availability of innovative and safe medical devices. For more information, visit the web site:

http://mdiss.org/

11.2 About Codenomicon

Codenomicon is a market leader in proactive IT security testing and real-time network and incident situation awareness solutions. For more information, visit the web site:

http://www.codenomicon.com/