Finding Software Vulnerabilities

Tommy Thorsen

10th January 2017

As computers grow increasingly powerful, they become able to run a mind-boggling amount of code in the blink of an eye. Software developers are taking full advantage of this, and software packages too, are growing in both size and complexity. The amount of code that is run for the simple task of showing a web page, ranging from device drivers, operating system calls, network stack, html parser, javascript engine, etc, is simply staggering.

Software security follows the principle of the weakest link, meaning that if an attacker can find a single vulnerability in any of these software packages, he can compromise the system. With such an impossibly large attack surface, how can we make sure we are safe from attackers?

In this study, we will dive into the world of software vulnerabilities, and how to detect them. From manual code audits to fully automated code analysis tools, we will look at the most commonly used techniques for finding flaws. We will look at methods used by software developers to find vulnerabilities in their own code, to methods used by attackers to find flaws to exploit.

1 Introduction

The ability to discover vulnerabilities in software is something that is useful to anyone in the computer industry. Whether you are a black-hat hacker looking for the next big heist, a security researcher looking to protect your clients, or just a regular software developer wanting to make sure your code is safe, you need to know how to find vulnerabilities.

When we are looking for vulnerabilities, the first thing to look for is whether we are able to make the program crash. Crashes often indicate improper memory handling, and may be possible to exploit as some kind of overflow. After we've found a crash, our next step depends on our goals. If our goals are to find exploitable vulnerabilities, we need to actually start trying to create an exploit. Most often, this is done by means of a debugger and possibly by inspecting the program with a decompiler. If we are a software company, and we've found the crash in our own program, as part of a testing

process, we are probably not interested in an exploit, but we'd rather focus on creating a patch for the problem.

> *Exploiting a program is simply a clever way of getting the computer to do what you want it to do, even if the currently running program was designed to prevent that action. Since a program can really only do what it's designed to do, the security holes are actually flaws or oversights in the design of the program or the environment the program is running in. It takes a creative mind to find these holes and to write programs that compensate for them. Sometimes these holes are the products of relatively obvious programmer errors, but there are some less obvious errors that have given birth to more complex exploit techniques that can be applied in many different places.*
>
> —[Erickson, 2008]

There are many ways to go about finding crashes, exploitable or otherwise, and the methods can be split into *automated methods* and *manual methods*. The automated methods are very well suited to quickly scan through large amounts of code, and can be integrated into your source control system, in order to automatically catch security problems in newly written code. While these automated methods are very powerful, they are probably not going to be able to catch every last flaw in your code base. Sometimes you need a good old-fashioned human being, reading through the code. The manual discovery methods, although they require more effort to do, are still widely used.

1.1 Automated discovery methods

Fuzzing One of the most well known methods for finding software vulnerabilities is called fuzzing. The idea is that we can feed a stream of random values as input to a program, in order to discover if there are any values that cause the program to crash. A crash means that the program did not do a good enough job in validating the input, and if we do find a crash, we can go on to investigate if it is possible to exploit the fault.

Static analysis This is primarily useful if we have access to the source code, but can also be used with disassembled binaries. There exists a plethora of tools that will analyse your program for common coding mistakes and unsafe input handling. These are often used by developers to double-check their own code, but they can just as well be used by black-hats to search for their next target.

Dynamic analysis There are also some analysis tools that do not look at the static code, but analyse the code as it is being run. The most famous of these tools might be Valgrind. Tools like these are able to find some memory management issues that are really difficult to see by just looking at the static source code.

1.2 Manual discovery methods

Code auditing Many software projects do these as commit reviews prior to merging new changes into the code base. Some software projects also have continuously running manual auditing processes — one of the more famous ones being OpenBSD's audit process [BSD, 2015].

Reverse engineering The most time-consuming, and difficult of the vulnerability discovery methods involve taking a program for which you have no source code, and reverse engineering it from binary/assembly to something readable.

We will dedicate a section to each of these methods below, where we will go into a lot more details, and look at what research and tools exist.

2 Fuzzing

2.1 The history of fuzzing

The term *fuzzing* or *fuzz testing* is likely to have been coined by Barton Miller at the University of Wisconsin around 1988 [Miller, 2008]. The word *fuzz* was chosen to represent the random noise-like behaviour observed when trying to type commands over a dial-up modem line during a heavy thunderstorm.

> *I was logged on to the Unix system in my office via a dial-up phone line over a 1200 baud modem. With the heavy rain, there was noise on the line and that noise was interfering with my ability to type sensible commands to the shell and programs that I was running. It was a race to type an input line before the noise overwhelmed the command. This fighting with the noisy phone line was not surprising. What did surprise me was the fact that the noise seemed to be causing programs to crash.*
>
> —[Miller, 2008]

In an attempt to use this discovery to perform tests on common Unix utilities, he set his students to the task of writing fuzz generators as a class project. These fuzz generators would generate streams of ascii characters to be fed into the programs being tested. One of the groups were able to cause a large percentage of the utilities to crash, using this method. They published one of the first scientific papers on the newborn art of fuzzing [Miller et al., 1990].

2.2 Modern use of fuzzing

Since Miller's original work, a lot of research has been done and a multitude of papers have been published on the subject of fuzzing. While Miller's original fuzzing tools only concerned themselves with ascii input, the art of fuzzing has grown to encompass all

kinds of inputs. In addition to fuzzing simple data types like strings and numbers, it's common to fuzz entire file types or entire protocols.

If you want to test something like a web browser or a video decoder, it might not work very well to send completely random data. The vast majority of the time, parsing or decoding completely random data would cause a parsing/decoding error really early, and we wouldn't be able to exercise a very large amount of the target code.

> *Although fuzz testing can be remarkably effective, the limitations of black-box testing approaches are wellknown. For instance, the* `then` *branch of the conditional statement* `"if (x==10) then"` *has only one in 232 chances of being exercised if x is a randomly chosen 32-bit input value. This intuitively explains why random testing usually provides low code coverage.*
>
> —[Godefroid et al., 2008]

Instead we need to pass a file that's structurally more or less correct, but has some random or extreme elements. One popular way to do this is to start with a valid file, and make random changes to it before handing it to our target program [Sutton and Greene, 2005]. This method can be modified to work for other things such as network traffic. For instance, we may record a network transaction, run all the network packets through a fuzzing tool, and then replay the transaction.

Although we talk about inserting random values, most fuzzers don't only use random values. Often we will have a *fuzz vector* containing a set of particularily extreme or evil values. The contents of the fuzz vector can depend on what we want to fuzz, but may contain things like very large positive or negative numbers, 0, very long strings, strings with unicode, empty strings, strings containing null-characters etc.

Fuzzers that modify existing good data are sometimes called *mutation fuzzers* whereas fuzzers that generate new random (or semi-random) data are called *generation fuzzers* [DeMott, 2006]. Generation fuzzers do not have to generate completely random data — they are often hand-written to fuzz a specific file type or protocol type, and can generate structured data of varying degrees of validness.

Another common way to separate fuzzers and fuzzing methods, is by how much information they have about the target program. We will look at the three types of fuzzers, *black-box*, *white-box* and *gray-box* fuzzers in the following sections.

2.3 Black-box fuzzing

The term *black-box fuzzing* means that the fuzzer does not know anything about the internals of the target program. The only thing we know is the requirements of the functionality of the program, and any publicly documented interfaces it has. If we are fuzzing a program that we do not have source code for, this is the only possibility open to us. Hackers typically have to resort to black-box fuzzing, as they are looking for vulnerabilities in someone else's code.

2.3.1 SPIKE

The SPIKE fuzzer, detailed in [Aitel, 2002], is one of the earlier tools for use in black-box fuzzing. The paper speaks a lot about a revolutionary block-based approach to fuzzing. What Aitel means by block-based, is that the tool makes it easy to insert fields that contain the length of a block of bytes. This is a pattern that can be found in most file formats and protocols. SPIKE is a scriptable tool, and to use this block feature, we can define a block with a corresponding length field in SPIKE script, like this:

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata");
s_binary("01020304");
s_block_end("somepacketdata");
```

The code above will create a file that contains a big-endian length field that contains the length of the binary blob that follows. In order to fuzz an HTTP GET request, we could use something like the following SPIKE script:

```
s_string_variable("GET");
s_string_variable(" ");
s_string_variable("/");
s_string_variable(" ");
s_string_variable("HTTP/1.1\r\n");
```

This use of scripting, combined with the block feature, makes SPIKE a very powerful tool for quickly developing a fuzzing strategy for any given target. Although it's more than a decade old, it's still in use today.

2.4 White-box fuzzing

White-box fuzzing refers to the fuzzing of programs for which we know a lot about the implementation. If a company is doing internal testing on its own product, or we are fuzzing for vulnerabilities in an open source project, we have full access to all the source code. Since we are also able to build such software, we have access to symbols and runtime debugging information. The person doing the fuzzing may also be intimately familiar with the program and its source. This can be of great help to us when we are making or configuring the fuzzer, as we can selectively target areas we think are weak. Since we know sizes and boundary values of internal buffers, this also helps us select strategic fuzz vectors.

2.4.1 SAGE

[Godefroid et al., 2008] discusses white-box fuzzing in the context of internal product testing at Microsoft. It also describes the creation of a fuzzing tool called SAGE. Microsoft had at the time been using both black-box testing and static code analysis (which we will describe later) as part of their testing process, but they were still seeing that serious flaws made it into published software.

Notably, without any format-specific knowledge, SAGE detects the MS07-017 ANI vulnerability, which was missed by extensive blackbox fuzzing and static analysis tools. Furthermore, while still in an early stage of development, SAGE has already discovered 30+ new bugs in large shipped Windows applications including image processors, media players, and file decoders. Several of these bugs are potentially exploitable memory access violations.

—[Godefroid et al., 2008]

The SAGE tool uses a technique for *dynamic test generation*, which involves symbolically executing the program while observing all conditional statements encountered along the way, and recording these statements as constraints. The program is then re-run, with some or all of the constraints negated in order to trigger a new execution flow. This ensures that as many code paths as possible are tested. SAGE assumes that the input to the program is in the form of a file, and given an initial file, it can make modifications to that file that generate new tests. Interestingly, it's not always necessary to give SAGE a valid file — in one case, when given a file that was all zeroes, SAGE was able to generate a suite of test cases, deducing much the file format from the conditional statements in the code.

The results were very positive, as SAGE was able to find both known issues in older, unpatched binaries, and new, undiscovered issues in production software.

2.4.2 BuzzFuzz

Like the SAGE tool, the BuzzFuzz tool [Ganesh et al., 2009] aims to solve the problem of black-box fuzzing not being able to exercise significant portions of the program flow. Many of the concepts are the same, but BuzzFuzz uses a slightly different strategy. While SAGE treats all of the code the same, BuzzFuzz has the concept of *attack points*, which are pieces of code that have a higher risk of errors. The paper mentions library and system calls as two such attack points. The theory is that these calls are typically between different codebases, written by different developers, which causes a higher risk of problems. These are also areas that are typically not tested well enough with traditional testing methods.

The paper also mentions the concept of *taint tracing* which is an important concept in dynamic test case generation and white-box fuzzing. Taint tracing is a special instrumented execution method that keeps track of every variable that has been influenced by the input. For each variable, it is important to know which bytes of the input affects it, so that the input can be fuzzed accordingly.

In order to compare the performance of BuzzFuzz with normal black-box fuzzing, the team also wrote a simple fuzzer that would only make completely random changes to the input files. The results are very interesting, as they do not show that one approach is better than the other.

In comparison with other testing approaches, random testing offers clear advantages in automation, ease of use, and its ability to generate inputs that

> *step outside the developer's expected input space. Our new directed fuzz testing technique, as implemented in our BuzzFuzz tool, complements existing random testing techniques to enable, for the first time, fully automatic generation of test inputs that exercise code deep within the semantic core of programs with complex input file formats.*
>
> —[Ganesh et al., 2009]

Since the random, black-box fuzzing is a much simpler algorithm, it is able to run many more tests than the BuzzFuzz can in the same time-span. When looking at errors found per hour, the two tools perform at a similar level, but they are able to find different errors. This indicates that we're not dealing with two competing strategies, but rather with two complimentary algorithms.

2.5 Gray-box fuzzing

As we've seen, there is no clear winner between black-box and gray-box fuzzing — maybe the answer lies somewhere in between? The term *gray-box fuzzing* refers to an approach that is a mix of white-box and black-box algorithms.

2.5.1 LynxFuzzer

Described in [Mazzone et al., 2014], the LynxFuzzer was written to test kernel extensions in Mac OS X. The reason it can be categorized as gray-box, is that it contains several different fuzzing algorithms; both black-box and white-box. The following are the key components of the LynxFuzzer:

Tracer The tracer's job is to extract the so-called *dispatch table* from the kernel extension. The dispatch table is a special construction in Mac OS X, which is used for communicating between the user and the kernel. The dispatch table can be parsed to figure out reasonable bounds for later fuzzing.

Sniffer The sniffer observes live calls to the kernel extension and extracts the function being called, and the parameters being passed. This will also serve as basis for the fuzzing algorithm later.

Fuzzer The fuzzer consists of three different internal fuzzing engines. The generation engine uses the information from the dispatch table to generate valid but randomized calls to the various functions of the kernel extension. The mutation engine replays fuzzed versions of the live calls caught by the sniffer. And finally, the evolution engine leverages concepts of evolutional algorithms to learn from previous attempts and find better test cases with more code coverage.

All of these run on top of a hypervisor component which leverages the HyperDbg hardware-assisted virtualization framework. This allows LynxFuzzer to intercept calls to strategic methods and inspect memory in order to extract necessary data.

> *We implemented and evaluated LynxFuzzer on Mac OS X Mountain Lion and we obtained unexpected results: we indivuated 6 bugs in 17 kernel extensions we tested, thus proving the usefulness and effectiveness of our framework.*
>
> —[Mazzone et al., 2014]

It seems that by combining elements of both white-box and black-box fuzzing, the creators of LynxFuzzer were able to create a tool that could uncover as yet unpatched vulnerabilities in the Mac OS X kernel. Although none of these issues turned out to be easily exploitable, the result is promising.

2.6 Final words on fuzzing

Fuzzing is by no means an exact science, and some proponents of more traditional testing methods express disdain, especially for the relatively crude art of black-box fuzzing. However, fuzzing has shown time and again that it is able to find a lot of flaws that traditional testing overlooks. Although it would be ill advised for a business that makes its living by producing software, to replace their QA department with a fuzzing tool, it is possible that running the software through a simple fuzzer now and then, would be a valuable addition to their test process.

While black-box fuzzing often becomes too naïve and simple, its simplicity is also its strength. Not a lot of effort is required in creating a custom generation fuzzer, or configuring a mutation fuzzer to run a particular test. White-box fuzzing has some amazing possibilities, but is also hard to implement, and presumably also hard to use efficiently. Many modern fuzzers tend to pick a middle road between the two, as a gray-box fuzzing strategy.

3 Static analysis

Most people who has worked with software development for some time, have probably at some point encountered a static code analysis tool. These tools typically run through all the source code and output a list of code lines that they deem risky or erroneous. While they typically have a lot of false positives, they are also usually able to find a good amount of real problems.

Since these tools parse and inspect the source code, they are usually limited to supporting a single or a few programming languages. This also means that we can only use this technique to find vulnerabilities in programs that we have the source for. This limits us to open source software, programs for which the source code has been leaked, and software that we are writing ourselves.

A large amount of static analysis tools exist, ranging from commercial solutions like the well-known Coverity Code Advisor to open-source tools such as `cpplint`. The first static code checker may have been the `lint` tool, which was published for Unix in 1979 [Johnson, 1977].

Static code analysis tools may be used to detect a range of common programming mistakes. They can be used to find memory leaks and other allocation problems. Null-pointer dereferencing problems, uninitialized variables, missing return statements, logical errors etc.

3.1 Comparing popular tools

[Emanuelsson and Nilsson, 2008] is a practical comparison between three of the most popular static analysis tools in 2006/2007, namely Coverity Prevent, Klocwork K7 and PolySpace Verifier/Desktop.

> *It is clear that the efficiency and quality of static analysis tools have reached a maturity level were static analysis is not only becoming a viable complement to software testing but is in fact a required step in the quality assurance of certain types of applications. There are many examples where static analysis has discovered serious defects and vulnerabilities that would have been very hard to find using ordinary testing; the most striking example is perhaps the Scan Project which is a collaboration between Stanford and Coverity that started in March, 2006 and has reported on more than 7,000 defects in a large number of open-source projects (e.g. Apache, Firebird, FreeBSD/Linux, Samba) during the first 18 months.*
>
> —[Emanuelsson and Nilsson, 2008]

The paper shows that the Coverity tool and the Klocwork tool are similar in functionality and performance. They are both tuned to give as few false positives as possible, although this increases the risk of false negatives, where real bugs are not reported by the tool. The PolySpace tool, on the other hand, does not want to risk false negatives, and as a result returns a high amount of false positives. For a development situation where one uses the tool from day one, this may be acceptable, but running the tool for the first time on a large codebase is likely to produce more false positives than anyone would be prepared to go through.

The technology behind the PolySpace tool seems to be more advanced, and it is able to find some complex problems that the other two are not able to. A downside to this, is that the tool becomes very slow when the codebase grows too large. The other two tools seem to have picked a better balance between correctness and efficiency. While the PolySpace tool looks very powerful on paper, it was deemed not usable in a real test scenario at Ericsson, because it took too long or produced too many false positives. The other two tools were run on a number of different projects with varying but positive results.

3.2 ITS4

Now that we have some general background on static analysis tools, it's time to check out some of the research on the subject. First out is [Viega et al., 2000], which describes

the tool ITS4 — a security-oriented static code scanner for C and C++.

The authors of this paper were motivated by the large amount of unsafe functions in standard libraries in C and C++, and by the lack of good, public information about programming securely. The ITS4 tool is intended for programmers and security auditors alike, and is meant to provide a better alternative than grepping for known problematic function names.

The two main goals was to have a very low rate of false negatives, and to be able to scan the whole source without missing things hidden behind preprocessor flags. The team therefore wrote a custom C/C++ parser that could be used instead of relying on the regular parser/preprocessor.

> *While [ITS4's] parsing model makes it poorly suited for highly accurate static analysis, the same model makes the tool very practical for real-world use; even with some facility for a heuristic-driven static analysis of the program, ITS4 can scan large programs efficiently, while still achieving adequate results.*
>
> —[Viega et al., 2000]

The result is a relatively simple tool that tokenizes the source code and looks for certain patterns from a vulnerability database, and flags risky parts of the code with a description of the problem and the estimated risk level. While this approach limits the tool's ability to analyse the more complex aspects of the code, it has the advantage of being fast and lightweight. While this may not be able to cover the testing needs of a software business, it does seem like a good approach for a quick vulnerability scan.

3.3 Splint

The authors behind [Evans and Larochelle, 2002] point out some of the same problems as we saw in the article about ITS4. Most vulnerabilities are caused by well-known and easily preventable problems. But since programmers keep doing the same mistakes over and over, due to either carelessness or lack of awareness about security, tools are required to catch these flaws before they make it into published software.

The proposed tool, called Splint, is also similar to ITS4 in some ways. It's a lightweight static analysis tool for C. Like ITS4, Splint is not concerned with absolute correctness, but takes a pragmatic approach targeting useful results and good performance. Unlike ITS4, Splint uses a real C parser, which allows it to reduce the amount of of false positives.

> *Splint finds potential vulnerabilities by checking to see that source code is consistent with the properties implied by annotations.*
>
> —[Evans and Larochelle, 2002]

The key feature of Splint is the introduction of annotations to functions and parameters. For instance, the annotation `/*@notnull@*/` indicates that the annotated parameter should never be NULL, and will output warnings for code that might cause this to happen.

Splint is also scriptable, so that it is possible to add new types of checks and warnings. This can be useful if we are looking for a particular type of vulnerability. The paper shows how we can use the built-in scripting functionality to add features such as taint-tracking, which is a concept we remember from the BuzzFuzz fuzzer. This can be used, for instance, to detect format string vulnerabilities.

While Splint seems powerful and versatile, there is significant work involved in annotating a code base the way the tool requires. If our goal is to find vulnerabilities in an unfamiliar code base, first adding these annotations is completely unfeasible. This means that Splint is primarily useful for checking one's own code, and mainly if one has been using Splint from the start of the project, and continually added annotations while writing the code.

3.4 VCCFinder

We now take the leap from the slightly older ITS4 and Splint tools, to recent research. [Perl et al., 2015] describes the VCCFinder tool. The term *VCC* stands for *vulnerability-contributing commit*, and unlike the previous tools, VCCFinder operates on individual commits to a code base, rather than on the whole repository at once. To be able to know what bad commits generally look like, the team performed a large database mapping of CVEs to commits in 66 different GitHub projects.

> *[B]ased on this database, we trained a SVM classifier to flag suspicious commits. Compared to Flawfinder, our approach reduces the amount of false alarms by over 99% at the same level of recall.*
>
> —[Perl et al., 2015]

Since this is a lightweight tool that does not do comprehensive analysis of the code base, it is natural to compare it to other lightweight tools like Flawfinder, Splint and ITS4. When the team ran both Flawfinder and VCCFinder on the same dataset, Flawfinder produced a staggering 5460 false positives for 53 true positives. When configuring VCCFinder with a precision that would reveal the same amount of true positives, it only returned 36 false positives. This is a very managable amount.

The VCCFinder tool differs quite a lot from other static analysis tools in that its primary focus is not on parsing the code, but on extracting other information from the commits. For instance, longer commits, submitted by new committers, in an area that has been changed back and forth a lot would be flagged as a suspicious commit, as history shows that such commits often cause vulnerabilities. A number of properties are calculated for each commit and fed into an SVM as learning data.

An SVM (support vector machine) is a machine learning algorithm that is very good at classifying objects into one of two groups. This SVM, after having been trained with the set of VCCs and the set of non-VCCs, is able to label a new commit as suspicious or not suspicious. A lot of interesting information can be found by looking at the resulting rules after the training is complete. In addition to the examples above, it was found that VCCs often contain large amounts of new code lines, but few deletions, which means that vulnerabilities are most often introduced when developing new features, and not when making smaller incremental edits.

> *Our experiments demonstrate that VCCFinder is able to automatically spot vulnerability-contributing commits with high precision; yet this alone does not ensure that an underlying vulnerability will be uncovered. Significant work and expertise is still necessary to audit commits for potential security flaws. However, our approach reduces the amount of code to inspect considerably and thus helps increase the effectiveness of code audits.*
>
> —[Perl et al., 2015]

In the end, the final tool was able to identify several suspicious commits that the authors proceeded to report to the owners of the projects following responsible disclosure practice. It is interesting and promising to see a tool that thinks outside the box, and succeeds in using a completely new approach to finding vulnerabilities. Since this tool is not very dependent on the actual programming language, it would also seem to be possible to use it for other languages than C/C++, provided enough training data can be found.

3.5 PacketGuardian

[Chen et al., 2015] describes a more specialised tool called PacketGuardian. This tool is developed for the sole purpose of verifying packet handling logic in network protocol implementations. Packet injection attacks constitute a common problem on the internet and network protocol implementations that are not extremely carefully implemented may be vulnerable to this.

PacketGuardian works on source code, and uses taint tracking to statically analyse the packet flow within the program. The primary defence against packet injection attacks from an off-path attacker in unencrypted communication, is the knowledge of certain protocol state that only the two main parties know about. It is therefore important to avoid leaking information about this state to a potential attacker. If the attacker manages to deduce this information, he will be able to successfully inject packets, which again enables him to inject payload or induce a denial of service.

Analysis is done in two steps. The first step, called *accept path analysis* analyses the code path required to get a packet accepted (i.e. a successful injection). The goal of this step is to identify the weakest accept path, which is the one with the least stringent state checks. The next step is the *protocol state leakage analysis*. This step uses static

taint analysis to find out how much information an attacker can glean about the protocol state with a single packet. Combining the results from these two steps gives us a number for how many packets an attacker needs to send to successfully inject a packet.

> *Different from previous work which reported vulnerabilities in TCP code base by manual inspection, our tool performs automated analysis, and outputs not only all existing ones but also 11 new highly exploitable ones.*
>
> —[Chen et al., 2015]

This paper shows that very specialised tools may be used to find certain problems that general static analysis tools are unable to find.

3.6 Final words on static analysis

While black-box fuzzing might be the tool of choice for hackers looking for the next vulnerability, it would not be wise to completely rule out the use of static analysis methods. A lot of software running critical systems are open source, and can be scanned with such tools. Challenges to overcome are the large amount of false positives in some tools, and the fact that only a few of the issues reported by traditional static analysis tools are actually exploitable. However, if one takes the time to familiarize oneself with such a tool, it may be possible to configure it so that it only outputs errors that are interesting in a vulnerability perspective.

We've seen that beyond the large, well-known corporate code analysers, there are smaller, more specialised static analysers that may work better for us. Since we are only interested in exploitable vulnerabilities, and not in general coding style errors, these specialised tools may give us more relevant results with fewer false positives.

> *No tool will eliminate all security risks, but lightweight static analysis should be part of the development process for security-sensitive applications.*
>
> —[Evans and Larochelle, 2002]

In a software development perspective, these tools are very relevant. If a static code analyser is adopted early on in a project, and used continuously from that point on, it is much more feasible to deal with the false positives, than if it is introduced at a later stage. Tools like VCCFinder, which analyses single commits, are also well suited for integration into a source control review system.

4 Dynamic analysis

Where static analysis examines the program when it's sitting still, dynamic analysis examines the behaviour of a running program. Well-known examples are tools like the open source Valgrind and Rational Software's commercial tool called Purify. Since

these tools are observing the program as it's running, they don't have to make any speculations about what may be unsafe. They can simply see what the program is doing wrong. Dynamic analysis tools are particularily good at finding memory handling bugs such as dereferencing of uninitialised pointers, double frees, out-of-bounds reading and writing etc. We will, as usual, take a closer look at a couple of relevant examples.

4.1 Valgrind

The tool Valgrind is arguably the most famous of all dynamic code analysis tools. It is explained in great detail in [Nethercote and Seward, 2007]. The paper points out that Valgrind is not just a tool, but in fact a dynamic binary instrumentation (DBI) framework which can be used to build dynamic binary analysis (DBA) tools. The command line tool `valgrind` that many of us are familiar with, has an option `--tool` which lets us choose which of the Valgrind tools to use. The default tool, called *Memcheck*, is the tool most people think of when they hear the word Valgrind.

One of the more useful uses of the Valgrind framework is to create *shadow value tools*. These are tools that keep metadata about every single memory location or register used by a program as a shadow state. This is what Memcheck uses to detect illegal memory usage in programs. By flagging memory locations as uninitialized or deallocated, Memcheck can break to the debugger if the program tries to access this memory.

> *Shadow value tools are powerful, but difficult to implement. Most existing ones have slow-down factors of 10x–100x or even more, which is high but bearable if they are sufficiently useful.*
>
> —[Nethercote and Seward, 2007]

The method Valgrind uses to instrument binary code is called *disassemble-and-resynthesise*. The program under analysis is first disassembled, then instrumentation code is inserted into the disassembled code, before the code is again compiled to machine code with a just-in-time compiler. This is a relatively slow process, but gives Valgrind a lot of power to observe the running program.

The key goals of the Valgrind developers are to make a framework that makes tool-writing as simple as possible, and that the resulting tools are as robust as possible. It does this sacrificing some speed in comparison with other DBI frameworks, such as Pin [Luk et al., 2005].

From our perspective, in our search for tools to discover vulnerabilities, the pre-made tools that come bundled with Valgrind have a lot to offer. The default Memcheck tool is an obvious candidate, with its ability to detect a wide range of memory-related problems. Other promising Valgrind tools are:

SGCheck A tool for finding stack and global array overrun errors. This tool can detect some memory errors that Memcheck can not, at the cost of some false positives.

DHAT A tool for examining memory usage patterns of programs.

Helgrind A thread error detector that might help us find exploitable race conditions.

DRD Another thread error detector.

Although it would require significant effort, we should also not dismiss the idea of writing our own vulnerability checker on top of the Valgrind core, or on top of one of the competing DBI frameworks like Pin or DynamoRIO.

4.2 Flayer

[Drewry and Ormandy, 2007] describes a tool called Flayer, which is built on the Valgrind framework and based on functionality from the Memcheck tool. It uses the technique of data tainting, which we've come across a few times already, to analyse data flow in a program. Every bit of data that comes from stdin, from a file or from a network source, is marked as tainted, so that we can observe how the tainted values propagate through the program.

> *Flayer finds errors in real software. In the past year, its use has yielded the expedient discovery of flaws in security critical software including OpenSSH and OpenSSL.*
>
> —[Drewry and Ormandy, 2007]

Flayer combines methods from different families of tools in order to find errors more efficiently. While it uses dynamic binary instrumentation as its main weapon, it also has the ability to redirect code flow into seldom-used execution paths by manipulating the outcome of conditional jumps — a technique we remember from white-box fuzzers like SAGE. Flayer also contains random fuzzing capabilities. The combination of Flayer's flow altering functionality and a black-box fuzzer, makes it possible to fuzz parts of the program that are in execution paths that are hard to reach.

Flayer also provides support for *guided source code auditing*. An interactive version of Flayer, called FlayerSh can be used to flay a program built with debugging symbols. This gives the auditor the possibility to rapidly follow tainted values through the program flow by viewing the source code for all locations where the values are used.

Another useful use case is to run two parallel instances of FlayerSh — one on an unpatched program, and one on a program containing a security patch — in order to compare two versions of the program. Flayer can be used to force the code flow through the path that leads to the vulnerable code, in order to test the validity of the patch.

4.3 DiSL

[Marek et al., 2015] describes the implementation of a new dynamic program analysis framework, called DiSL. The DiSL framework targets the Java platform, and operates on Java bytecode. Tools using DiSL can be implemented in Java, making heavy use of class and method annotations. A simple method execution time profiler can be implemented with the following Java snippet:

```
public class SimpleProfiler {
    @SyntheticLocal
    static long entryTime;

    @Before(marker=BodyMarker.class)
    static void onMethodEntry() {
        entryTime = System.nanoTime();
    }

    @After(marker=BodyMarker.class)
    static void onMethodExit() {
        System.out.println("Method duration " + (System.nanoTime() - entryTime));
    }
}
```

In addition to intercepting method entries and exits one can use other marker classes to intercept events such as the entry and exit of basic blocks. DiSL contains a whole range of useful markers and standard functionality, plus the possibility to write new such classes, should the default ones not suffice.

DiSL interacts with the JVM running the target program through the *Java Virtual Machine Tool Interface* (JVMTI). Every time the JVM wants to load a class, the JVMTI agent intercepts the loading, and passes the class bytecode to DiSL. The DiSL framework then instruments the code according to the loaded snippets, and passes the instrumented code back to the JVM to load.

> *We believe that the unique combination of the high-level programming model with the flexibility and detailed control of low-level bytecode instrumentation makes DiSL a valuable tool that can reduce the effort needed for developing new dynamic analysis tools and similar software applications running in the JVM.*
>
> —[Marek et al., 2015]

The DiSL framework is the latest development step in a series of attempts to build a powerful and elegant dynamic analysis framework for Java. It provides a very modern and concise way to do very low-level things, and does seem suited to writing quick tools to analyse for specific vulnerabilities in Java programs.

4.4 Final words on dynamic analysis

As we have seen, dynamic analysis lets us step away from the theoretical approach of static analysers, and look at what's actually, definitely happening when a program runs. One disadvantage of that, is that we only get to test the code paths that we are actually running, but tools like Flayer can be used to remedy this disadvantage.

Although dynamic analysis tools are easier to work with if we have the program's source code available, we've seen that binary instrumentation tools work just as well with closed-source programs.

5 Code auditing

The idea of code auditing is really a simple one — a person reads through all of the source code and flags any flaws he may find. In practice, however, it's not as simple as it sounds. First of all, it's very time-consuming. Secondly, the effectiveness of code auditing depends a lot on the person doing the auditing. Ideally, he should be a security expert, who is also an experienced code auditor. It takes a trained eye and a lot of experience to spot the more subtle security flaws.

Many software companies these days have mandatory code reviews for all code that goes into their final products. These code reviews often double as security audits, except that they are most often performed by fellow programmers, who are most often not sufficiently security-conscious.

For a malicous black-hat, manual code-auditing may also be a viable strategy for finding vulnerabilities. If the software he wishes to attack is open source, or the source code has been leaked, reading all of the relevant code may be the simplest way to find exploitable vulnerabilities. If the source code is not available, he may first have to reverse engineer the program before he can do the audit. We will discuss reverse engineering in the next section, after we've taken a closer look at some scientific methodologies for doing a sound source code review.

5.1 "Practical Code Auditing"

As source code auditing is primarily a manual process that involves experience and expertise rather than specific technologies, not a lot has been written on the subject. [Grenier, 2002] is a brief document that describes common coding errors and flaws, and what to look for in a code audit.

The paper suggests to do a code audit in multiple passes, first looking for things that are very obvious and easy to find, such as the use of known unsafe functions in the C library. This can be done with `grep` or with a small `perl` script. C functions such as `strcat` and `strcpy` are not allowed at all, and should be replaced with `strncat` and `strncpy`. Existing calls to `strncat` or `strncpy` should be checked for off-by-one errors.

> *Any usage of gets should be an immediate clue that our program is vulnerable, not only at that point, but probably many others. It should be quickly replaced with a sane buffered and checked input loop, or exploited, depending on your purpose.*
>
> —[Grenier, 2002]

The paper goes on to explain how the various types of memory overflows work, and how to spot a format string vulnerability. Integer overflows are explained with the following illustrative example:

> *For example, lets assume that we get from the network a number of structs we are to recieve. so*
>
> ```
> u_int num_of_structs = grab_from_input();
> ```
>
> *And so we naturally allocate memory next to hold this array of structs, like so*
>
> ```
> mystruct *mem = malloc(num_of_structs * sizeof(mystruct));
> ```
>
> *While both num_of_structs, and sizeof(mystruct) may be less than UINT_MAX, (and we can even check both to be sure they are individually under UINT_MAX), multiplying them together may result in a size which overflows, resulting in a small positive int. We have then created a buffer overflow when we try to copy into mem, even if we use safe copy functions.*

Binary auditing — the auditing of programs for which we have no source code — is also mentioned along with various techniques for testing the security of programs using *stress testing* and *fault injection*.

5.2 "Policies and Proofs for Code Auditing"

[Whitehead et al., 2007] takes code auditing to a whole new level with its BCIC language for proving the trustedness of a system. Rather than relying on best practices and general policies for auditing, this methodology aims to express the auditing process and requirements in a logical language. It's a very theoretical and academical paper, and it's hard to envision a black-hat looking for vulnerabilities, using an approach like this. It may however, have some merit for a software company to enforce a methodology like this for particularly security-sensitive components.

> *In this paper, using BCIC, we suggest an approach to code auditing that bases auditing decisions on logical policies and tools. Specifically, we suggest that policies for auditing may be expressed in BCIC and evaluated by logical means.*
>
> —[Whitehead et al., 2007]

One interesting suggestion in this paper, is the introduction of secure and insecure data types to the programming language. By making data from unknown outside parties have the *distrusted* type, and our internal functions requiring the *trusted* type, a tool can flag the cast operations between these two types as requiring an audit. Once a

trusted auditor has verified and signed the cast operations, the function can be analysed and a proof of trustedness can be generated.

This separation of the trusted and untrusted data types is reminiscent of the taint tracing methods we've seen earlier, except that in this case we are monitoring taint manually, at development-time.

6 Reverse engineering

Reverse engineering in this context, is the act of figuring out how a program works, by looking at its binary code. A good first step in reverse engineering a program for which you have no source code, is to run it through a *disassembler*, like IDA Pro or Hopper. This will translate the binary into assembly code, which is quite a bit more readable than the binary file, although not as readable as a higher level language.

> *The term "reverse engineering" has its origin in the analysis of hardware — where the practice of deciphering designs from finished products is commonplace. Reverse engineering is regularly applied to improve your own products, as well as to analyze a competitor's products or those of an adversary in a military or nationalsecurity situation.*
>
> —[Chikofsky and Cross, 1990]

Reverse engineering can be used as a way to find a vulnerability, or it can be used to verify whether a found flaw is exploitable. Let's say you've run a fuzzer on a program for which you have no source code, and you've found a crash. At this point, you will need to have a look at the assembly code for the program, in order to figure out what causes the crash and whether it can be used to cause for instance a stack overflow. In addition to the disassembler, you are most likely going to need a debugger, like gdb or visual studio.

Some times you may even want to skip other means of locating vulnerabilities and go straight to the reverse engineering. If the program is not prohibitably large this may be a viable strategy, but for most realistic programs, this will be too much work. Nevertheless, reverse engineering is a key technique to familiarise ourselves with, if we want to really understand vulnerabilities and exploits.

6.1 Taxonomy

[Chikofsky and Cross, 1990] is an early taxonomy of reverse engineering and related techniques. It provides the following definitions:

Forward engineering Regular engineering and development. We start with information on a high abstraction level and move towards an implementation.

Reverse engineering This is the process of analysing a subject system in order to create a representation of the implementation at a higher abstraction level.

Redocumentation A subarea of reverse engineering. Redocumentation is the process of creating descriptions of the subject system at roughly the same abstraction level.

Design recovery Another subarea of reverse engineering. By using deduction and domain knowledge, we attempt to create a higher-level description of the subject system.

Restructuring The transformation from one representation to another at roughly the same abstraction level. Often restructurings are done in order to improve code quality or robustness.

Reengineering Involves examining and altering a subject system in order to recreate it in a new form. Reengineering can be performed by doing reverse engineering followed by forward engineering.

The categories that are the most interesting to us, in our quest to find vulnerabilities, would be the various forms of *reverse engineering*. The process of *redocumentation* is commonly used when doing binary auditing.

6.2 Exploring reverse engineering

The study [Treude et al., 2011] is a qualitative study where seven engineers, working with reverse engineering of malware, were interviewed about their tools and work methodologies.

Reading assembly code, and documenting the code and data flow as they figured out how it worked, was a core task for all of the engineers. The lack of conventions, and lack of adherence to the few conventions that does exist, was identified as a problem, especially when it came to taking over someone else's work. The sharing of information and knowledge between teams and even between individuals within a team was also problematic due to constraints in tools and work methodologies. Some of these constraints were intentional and self-imposed, such as air-gapping of workstations where malware was being studied.

> *[T]he main analysis tool used by reverse engineers in the security context is "brain power"*
>
> —Interview subject no. 6, [Treude et al., 2011]

Most of the engineers reported using IDA Pro for dissasembly. IDA Pro is widely regarded as a very powerful and professional tool, although the price-tag tends to be a showstopper for amateur analysts. Documentation tended to be written on paper or using regular office tools. UML was in use, but finding good tools for drawing and navigating UML diagrams was a challenge. Communication was done via email and

chat, but most of the time, teams would be situated at the same place, allowing for face-to-face communication.

> *Right now it's being done like a craft, and we'd like to have some kind of assembly line*
>
> —Interview subject no. 4, [Treude et al., 2011]

The main learning we can take from this study, is that good tools for documenting and articulating the reverse engineering process do not exist, and many man-hours are wasted manually translating between different formats and detail levels.

6.3 Side channel disassembly

We have on a couple of occasions mentioned how it's easier to analyse the source code for a program than analysing the compiled binary program. There are, however, some cases where you don't even have a binary to access. The program could be embedded into a chip inside an electronic device. [Eisenbarth et al., 2010] suggests a way to use passive methods such as power or electromagnetic emanation measurements to reverse engineer the program running on a microcontroller.

By observing power usage signature of different combinations of instructions, the researchers could build a model that could make very good guesses as to which instructions were being run on the microcontroller.

> *Hence, we can positively assure that side channel based code reverse engineering is more than just a theoretic possibility. Applying the presented methodology allows for building a side channel disassembler that will be a helpful tool in many areas of reverse engineering for embedded systems.*
>
> —[Eisenbarth et al., 2010]

Some instructions turned out to have very similar power consuption signatures. This was solved by doing frequency analysis on instructions and instruction-pairs from a range of different programs. The resulting frequency tables could be consulted in order to further increase the accuracy of detection. The final side channel disassembler implementation showed a detection rate of up to 70%.

7 Conclusion

We have seen that there are many ways we can conduct our search for vulnerabilities. Even within the different categories, there are a multitude of different tools and methodologies. Which tool or process works best for you, depends on many factors; what kind of vulnerability you are looking for, whether or not you have the source for the program, what expertise and experience you have, etc.

The different categories of tools have different strengths and weaknesses, and it's impossible to say which one is the best approach. For most people, a combination of tools of different types will give the best results. Different tools and methods can complement each other, and using one tool to cover up for the weaknesses of another might be a sound strategy. Which tools and methods you end up using, is up to you and to your preference, but everyone who is serious about vulnerability-spotting, should aim to build themselves a toolbox of skills, tricks and tools that in sum lets them make the most of their abilities.

> *Like it or not, vulnerabilities exist in the software and networks that the world depends on from day to day. It's simply an inevitable result of the fast pace of software development. New software is often successful at first, even if there are vulnerabilities. This success means money, which attracts criminals who learn how to exploit these vulnerabilities for financial gain. This seems like it would be an endless downward spiral, but fortunately, all the people finding the vulnerabilities in software are not just profit-driven, malicious criminals.*
>
> —[Erickson, 2008]

References

[BSD, 2015] (2015). Openbsd security. Technical report, The OpenBSD team. http://www.openbsd.org/security.html, Accessed December 13th, 2015.

[Aitel, 2002] Aitel, D. (2002). The advantages of block-based protocol analysis for security testing. *Immunity Inc., February*, 105:106.

[Chen et al., 2015] Chen, Q. A., Qian, Z., Jia, Y. J., Shao, Y., and Mao, Z. M. (2015). Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 388–400.

[Chikofsky and Cross, 1990] Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17.

[DeMott, 2006] DeMott, J. (2006). The evolving art of fuzzing. *DEF CON*, 14.

[Drewry and Ormandy, 2007] Drewry, W. and Ormandy, T. (2007). Flayer: Exposing application internals. *WOOT*, 7:1–9.

[Eisenbarth et al., 2010] Eisenbarth, T., Paar, C., and Weghenkel, B. (2010). Building a side channel based disassembler. *Transactions on Computational Science*, 10:78–99.

[Emanuelsson and Nilsson, 2008] Emanuelsson, P. and Nilsson, U. (2008). A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21.

[Erickson, 2008] Erickson, J. (2008). *Hacking: the art of exploitation.* No Starch Press.

[Evans and Larochelle, 2002] Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *software, IEEE*, 19(1):42–51.

[Ganesh et al., 2009] Ganesh, V., Leek, T., and Rinard, M. (2009). Taint-based directed whitebox fuzzing. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 474–484. IEEE.

[Godefroid et al., 2008] Godefroid, P., Levin, M. Y., Molnar, D. A., et al. (2008). Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166.

[Grenier, 2002] Grenier, L. A. (2002). Practical code auditing.

[Johnson, 1977] Johnson, S. C. (1977). Lint, a c program checker.

[Luk et al., 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM.

[Marek et al., 2015] Marek, L., Zheng, Y., Ansaloni, D., Bulej, L., Sarimbekov, A., Binder, W., and Tuma, P. (2015). Introduction to dynamic program analysis with disl. *Sci. Comput. Program.*, 98:100–115.

[Mazzone et al., 2014] Mazzone, S. B., Pagnozzi, M., Fattori, A., Reina, A., Lanzi, A., and Bruschi, D. (2014). Improving mac OS X security through gray box fuzzing technique. In *Proceedings of the Seventh European Workshop on System Security, EuroSec 2014, April 13, 2014, Amsterdam, The Netherlands*, pages 2:1–2:6.

[Miller, 2008] Miller, B. (2008). Foreword for fuzz testing book. http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html, Accessed December 13th, 2015.

[Miller et al., 1990] Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44.

[Nethercote and Seward, 2007] Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM.

[Perl et al., 2015] Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., and Acar, Y. (2015). Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 426–437.

[Sutton and Greene, 2005] Sutton, M. and Greene, A. (2005). The art of file format fuzzing. In *Blackhat USA Conference*.

[Treude et al., 2011] Treude, C., Filho, F. M. F., Storey, M. D., and Salois, M. (2011). An exploratory study of software reverse engineering in a security context. In *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, pages 184–188.

[Viega et al., 2000] Viega, J., Bloch, J.-T., Kohno, Y., and McGraw, G. (2000). Its4: A static vulnerability scanner for c and c++ code. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, pages 257–267. IEEE.

[Whitehead et al., 2007] Whitehead, N., Johnson, J., and Abadi, M. (2007). Policies and proofs for code auditing. In *Automated Technology for Verification and Analysis*, pages 1–14. Springer.