Software Security Lecture Notes

Lecture 1: How software breaks (native)

Time: Wednesday 12 March 2014, 16:15–18:00

Place: Exactum, D122

Summary

In order to understand how to build secure software, one must first understand the ways in which software can be broken. The main focus during the first lecture is native code. We discuss software defects in native code which often lead to crashes, some of which potentially open up opportunities to gain unauthorized access to a computer system. As an example of such a defect, we discuss “old school” buffer overflow attacks, which pose such a significant risk that certain operating system-level countermeasures have become widespread. Such countermeasures, however, constitute only a last line of defense against would-be intruders. We also discuss fuzzing, a software testing technique suitable for detecting input related software defects in native code.

Content of lecture

The topic of the course is software security, i.e. building secure software, as opposed to building security software. The former involves software development process practices where the security of the system being built is taken into account during each phase of the development process, essentially building security in from the ground up, whereas the latter involves building software designed for enhancing information security in the environments they operate in.

In the US, penetration testing has become widespread to a point where the hourly costs of penetration testing are decreasing. However, penetration testing at the late stages of a software development project is not enough, and hence activities occurring early and repeatedly during a software development process, such as threat modeling and architectural threat analysis have recently seen an increase in value in the industry.

Terminology [9]

Weakness. Weaknesses are ways in which software breaks down. Common software weaknesses are catalogued by the Common Weakness Enumeration (CWE) initiative\(^1\), which provides a community-developed taxonomy of software weakness types, free for public use. The CWE initiative is managed by the MITRE Corporation, an
American not-for-profit organization based in Bedford, Massachusetts and McLean, Virginia. The MITRE Corporation operates multiple federally funded research and development centers in the USA.

**Vulnerability** Vulnerabilities are software defects which enable an attacker (see below) to compromise the security of a software system. A vulnerability consists of the susceptibility of a system to a particular weakness, access to the weakness, and the capability to exploit (see below) the weakness. Publicly known vulnerabilities are catalogued by the National Vulnerability Database (NVD)\(^1\), which uses the Common Vulnerabilities and Exposures (CVE)\(^2\) standard for information security vulnerability names.

Each vulnerability is assigned a CVE identifier, e.g. CVE-2014-0060. The CVE identifier uniquely identifies the vulnerability in question. Each CVE contains an overview of the vulnerability, a list of vulnerable software and versions, a classification on the impact of the vulnerability and potentially references to external advisories, solutions and tools related to the vulnerability, such as issue tracker entries, exploit code etc. The CVE List Master Copy and assignment of CVE identifiers is managed by the MITRE Corporation.

**Bug** A bug is a implementation level software defect. Software security concerns itself primarily which bugs which might cause vulnerabilities, e.g. buffer overflows, race conditions, insufficient input validation etc.

**Flaw** A flaw is a software defect at a much deeper level than bugs. A flaw can for instance be an issue in the logic, design or architecture of a software system.

The distinction between bug and flaw is not rigid. McGraw06 [5] uses software defect as a collective term encompassing both bugs and flaws.

**Exploit** The term exploit can be used both as a noun and a verb. When used as a noun, it refers to the actual program code that can be run to take advantage of a vulnerability. When used a verb, it refers to the act of compromising the security of a software system through a vulnerability.

**Risk** Risk is defined as the probability of a weakness existing in a particular system, combined with the potential impact of exploiting a vulnerability caused by the weakness. In a business context, the impact can be both tangible, e.g. direct monetary losses, as well as intangible, e.g. damage to the credibility of a business or product.

**Attack** The act of attempting to compromise the security of a system. The act of exploitation is one type of attack. The term attack does not by itself imply malicious

\(^1\)https://cwe.mitre.org
\(^2\)https://nvd.nist.gov
\(^3\)https://cve.mitre.org
intent. A security analyst can attack a system with the intent of finding vulnerabilities in order to disclose them to the maintainer of the system under attack. This can take the form of penetration testing during an security audit.

**Attacker** A neutral term identifying the party (person or system) performing an attack. A related term is adversary; one that contends with, opposes, or resists, i.e. an opposing force with conflicting interests. In context of software security, similarly to attacker, adversary doesn’t necessarily imply maliciousness.

**Threat** The exact meaning of the word threat varies in information security literature. For the purposes of this course, we consider a threat to be an actor that may exploit a vulnerability, or the event of exploitation. In other contexts, the term threat can refer to a physical person (see for instance McGraw06 [5]), while alternative interpretations also include using the term threat as a synonym for risk (this is done widely in literature from Microsoft).

**Threat Actor** The, possibly malicious, physical person behind a threat.

**Attack Surface** The attack surface is everything that an attacker can interact with or control, including input or state.

**Attack Vector** Conveyor of the attack, i.e. the means by which an attacker attempts to gain access to a system (cf. disease vector in biology, i.e. a carrier of a disease). Examples of attack vectors include network protocols, e-mail attachments, web-pages and even deception, in which social engineering is used to fool a human operator into circumventing security mechanisms.

**Overview of “Old School’ Buffer Overflow Exploits**

Buffer overflow attacks [6] target software defects which lead to improper or missing bounds checking on buffer operations. Buffer overflow attacks are typically triggered by input injected by an attacker. As a consequence of a successfully performed attack, an attacker is able to write past the boundaries of allocated buffer regions in memory, causing a program crash or potentially redirect the execution of program statements as per the attackers’ choice. Buffer overflow attacks can be devastating, potentially allowing arbitrary code execution with the privileges of the vulnerable software. Therefore, weaknesses which might lead to buffer overflow attacks are particularly interesting, and have lead to the development of various operating system level safeguards to prevent exploitation.

A computer program is a sequence of instructions which are executed one by one by the Central Processing Unit (CPU). In a multiprocessing operating system, each instance of a program being executed is called a process. In modern multiprocessing operating systems, each process is assigned a region of virtual memory, which includes the executable code, process-specific data, a call stack, and a heap to hold intermediate computation data during run time. A typical memory layout is shown in Figure 1. The operating system
also makes sure that processes cannot access the memory regions of other processes, with the exception of explicitly shared memory.

The binary code of the program is loaded into memory and placed in the text section of the process. Any shared libraries, regardless of whether they are statically or dynamically linked to the program, are also loaded in the process memory space, and execute with the same process privileges as the rest of program.

In the Von Neumann computer architecture, the CPU control unit contains a *Program Counter* [6] (PC), which (in most CPU architectures) contains the memory address of the next program instruction to be executed. This is how the CPU keeps track of program execution. Each time an instruction is fetched from memory, the PC value is incremented. In the x86 processor architecture, the PC is called the *Instruction Pointer* (IP), and is stored in the EIP register⁴.

---

⁴[https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture](https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture)
The stack consists of a number of *stack frames*, each corresponding to a call to a subroutine which has not yet terminated. The size of the stack if kept track of via a *Stack Pointer* (SP), which contains the lowermost memory address of the stack, i.e. the top of the downwards growing stack. Before a subroutine call occurs in the process, any arguments to the subroutine are *pushed* onto the stack by the caller, i.e. copied to the top of the stack indicated by the memory address stored in the SP. When the subroutine call occurs, the return address, i.e. the current IP value is likewise copied from the EIP register and pushed onto the stack. Then, the memory address which is the entry-point to the subroutine is loaded to the EIP, overwriting the previous IP value. During the execution of the subroutine, any local variables are stored on top of the stack. When the subroutine execution completes (as indicated by a *ret* instruction), the stack frame is freed, and the saved IP value is popped from the stack and loaded in the EIP. This causes program execution to proceed from the next instruction following the subroutine call.

In a buffer overflow attack [6], the goal of the attacker is to overwrite the previous IP value stored on the stack, thus causing execution after the return from the current subroutine to proceed from a point chosen by the attacker.

A buffer overflow may occur as a result of careless memory copy operations. Examples of such operations include improper use of the *strcpy()* function. The *strcpy()* function takes two parameters; dest; and src. The function copies the string pointed to by src, including the terminating null byte (\0), to the buffer pointed to by dest. A precondition mentioned in the function specification is that the destination buffer dest must be large enough to receive the copy. If this is not the case, and dest happens to be a local buffer allocated from the stack, *strcpy()* will proceed past the bounds of the buffer, potentially overwriting memory allocated for other local variables and eventually the return address, if the string pointed to by src is large enough. In such a situation, if an attacker can influence the content of src, e.g. *strcpy()* is used to copy user provided input to a local buffer, the program is vulnerable to a buffer overflow attack.

To exploit a buffer overflow vulnerability, the objective of the attacker is two-fold; firstly, to craft suitable input which will overflow the buffer, and result in the stored return address being overwritten by a memory address chosen by the attacker; secondly, to inject program instructions chosen by the attacker into the process memory space. This payload is called a *shellcode* [6], because it typically results in the invocation of a command shell the attacker can use to control the compromised host. The overwritten return address will be chosen by the attacker in such a way, that upon return from the current subroutine, the CPU will jump to the attacker's shellcode.

In its simplest form, the shellcode is included in the input and is written to the stack frame along with the return address chosen by the attacker. However, unless the attacker access to the memory layout of the process, he is faced with the problem of determining the exact address of the buffer in order to jump to the beginning of the shellcode. The oldest and most widely known technique to circumvent this issue is called the NOP-
Figure 2: NOP-sled buffer overflow exploit

The idea is to corrupt large sections of the overwritten memory with values corresponding to no-op machine instructions; then place the actual shellcode after the no-op instructions (see Figure 2). The sequence of no-op instructions is referred to as the NOP-slide, since if the attacker manages to overwrite the return address with any adress within the NOP-sled, program execution will “slide” up the corrupted section, eventually hitting the actual shellcode at the end. This mitigates the problem of determining the exact address of the overflown buffer by allowing the attacker to take a guess at a proper value for the return address which replaces the original return address. The NOP-sled increases the size of the valid target area which will lead to the execution of the shellcode.

The success of a NOP-sled attack depends on the attacker successfully guessing an offset within the NOP-sled. The chance of success depends on the size of the NOP-sled. The amount of memory available for the NOP-sled depends on the amount of memory allocated for the overflown buffer, and the current depth of the stack.

To overcome these shortcomings, a slightly more sophisticated technique can be employed to jump to the memory location pointed to by the SP upon subroutine return. This allows the attacker to avoid the need to guess stack offsets and a large NOP-sled. Instead, the attacker injects the shellcode relative to the SP into the memory occupied by possible call parameters and the previous stack frame (see Figure 3). To achieve this, the attacker must determine the address of a suitable trampoline [9] instruction somewhere in program memory, such as jmp %esp in the case of the x86 architecture, which in turn would transfer the flow of execution to the exploit instructions on top of the stack.

In practice, a particular program might not intentionally contain instructions to a jump to a particular register. However, since the x86 instructions are neither of fixed length, nor of fixed alignment, the attacker is likely to find an unintentional instance of a suitable opcode at some offset of instructions in program memory [8].
Consider for instance the following instruction, found at address 0x00172718:

```
0x00172718: 84 2d fb ff e4 16 test %ch,0x16e4fffb
```

While this instruction performs a bitwise AND on two operands, interpreted at offset 0x0017271b, the second operand contains 0xff 0xe4, the opcode for the jmp %esp instruction in the x86 instruction set.

```
0x0017271b: ff e4 jmp %esp
```

By contrast, on an architecture such as MIPS, all instructions are 32 bits long and 32-bit aligned, and hence there is no ambiguity regarding where instructions start or stop. On such architectures, unintentional instructions, as the one described above, do not occur [8].

**Buffer Overflow Exploit Countermeasures**

The first line of defense against buffer overflow exploits is to avoid buffer overflows altogether. Automatic protection at language level is the most significant development to this end. Software written in languages without built-in buffer overflow protection can benefit from static analysis tools, especially code security scanners. Even the earliest tools, such as ITS4, can indicate instance of the usual suspects, such as the invocation of `strcpy()` and `gets()`, which are commonly associated with buffer overflow vulnerabilities [5]. These development-time aids, however, cannot help with legacy code which exhibit buffer overflows.

Vulnerable legacy applications can in some cases be retrofitted to protect against the exploitation of buffer overflow exploits [3]. Examples of this include the AusCERT Overflow Wrapper⁷, which was developed as a workaround to the SGI IRIX login/scheme buffer overrun vulnerability. The vulnerability allowed local users to gain root privileges.
due to insufficient bounds checking on arguments supplied by users. The workaround could be applied in cases where the official vendor patches to address the vulnerability could not.

However, due to the serious nature of buffer overflow vulnerabilities, and the extent to which legacy software is vulnerable, a number of architectural countermeasures have been deployed [9]. Widespread countermeasure include, stack canaries, $W\oplus X$ memory protection and Address Space Layout Randomization (ASLR).

Stack Canaries

Like the proverbial “canary in a coal mine”†, stack canaries [2], or stack cookies provide a warning system in case of buffer overflow attacks. Stack canaries are typically implemented as a compiler feature, which allows programs to be augmented at compile time to perform stack protection.

Such augmentation involve placing a canary value next to the return address on the stack, as shown in Figure 4. Upon subroutine return, the canary value is verified to be intact before jumping to the address pointed to by the stored return address. The protection is based on the assumption that the return address is unaltered if, and only if, the canary value is unaltered. As buffer overflow attacks exploit the fact that the return address is close to a buffer with inadequate bounds checking, the attacker is usually restricted to performing linear, sequential write operations to memory in ascending order. Under such circumstances, it is very difficult to overwrite the return address word without disturbing the canary value.

†An allusion to caged canaries (birds) that mining workers would carry down into the mine tunnels with them. If dangerous gases such as methane or carbon monoxide leaked into the mine, the gases would...
An important factor with stack canaries is the choice of canary value. If the canary value is known to the attacker, it is possible to devise a payload which will overwrite the canary value with an identical sequence of bytes, then proceed to overwrite the return address with one chosen by the attacker. For this reason, one or more random canary values can be generated at program initialization, then used throughout the lifetime of the program. Unless the attacker has access to the memory image of the running program, guessing such canary values will be difficult, provided that the values are generated with sufficient entropy and are not predictable. However, in certain cases it might still be possible for an attacker to bypass the stack canary altogether [1], e.g. by overwriting the return address via a stray pointer pointing to it, if one exists. To guard against such circumstances, the recorded canary value can be determined by XOR'ing the random canary value and the return address, effectively encrypting the return address with the random canary value. This operation is repeated when the stored canary is validated upon subroutine return. If the attacker has managed to modify the return address, the XOR'd random canary will not match the recorded canary.

**W⊕X**

W⊕X prevents attackers from injecting their own shellcode payload onto the stack or heap and executing it by ensuring that protected program segments are not writable and executable at the same time. W⊕X is typically implemented using a non-executable bit that the hardware platform enforces. On the x86 architecture, the AMD No eXecute (NX), or Intel eXecute Disable (XD) bit can be set using the Physical Address Extension (PAE) addressing mode [7]. The ARM architecture supports a similar eXecute Never (XN) bit. If the next instruction to be executed is loaded from a memory page with the non-executable bit enabled, the hardware raises a fault.

Beginning with Microsoft Windows XP SP2, Windows supports hardware-enforced Data Execution Prevention (DEP) on hardware which supports either the NX or DX bit. However, DEP is enabled by default only for limited system binaries and programs that "opt-in". If hardware support is not available, software-enforced DEP is enabled only for a limited set of system libraries.

Linux on the x86 architecture supports W⊕X in kernel configurations with PAE enabled [7]. On older hardware, without PAE support, Linux kernels with the ExecShield feature, can emulate W⊕X using x86 segments [10].

The main drawback of W⊕X is that it only prevents attackers from injecting new code into memory. In a return-to-libc attack, an attacker replaces the return address on the call stack with an address of a function that is already loaded in the binary or via shared library. The C standard library, libc is the most likely target, as it is always linked to the program. An attacker can for instance invoke an arbitrary program via the `system()`

---


system call wrapper. We@X does not prevent return-to-libc attacks because the executed code is already loaded into the process memory, and is intended to be executable.

We@X was still considered worthwhile to deploy since return-to-libc was considered as a more limited attack compared to arbitrary code injection, since in a return-to-libc attack, the attacker is constrained to calling one function after another, thus executing only straight-line code, as opposed to branching and other arbitrary behavior available via code injection. Furthermore, an attacker was thought to be limited to only those functions available in the program’s text segment and loaded libraries, so by removing certain functions from linked libraries it might be possible to restrict the capabilities of the attacker.

However, in a more powerful variant of the return to libc attack, known as Return Orientated Programming (ROP) [8], an attacker overwrites the stack with a series of return addresses, each jumping back to short snippets of (possibly unintended) instructions, called gadgets in the main binary or one of the dynamically loaded libraries. Each such gadget ends in a ret instruction, allowing the execution of sequence to proceed as intended. Suitable gadgets can be identified via automated tools, such as ROPGadget

The x86 instruction set is particularly vulnerable to ROP, as a random byte stream can be interpreted as a series of valid instructions with high probability. In his 2007 paper detailing ROP-attacks, Hovav Scacham shows that the set of gadgets found via inspection in 32-bit libc is Turing complete, meaning return-orientated programs are capable of arbitrary code execution on the x86 architecture.

Address Space Layout Randomization

ASLR prevents an attacker from directly referring to known targets in memory by randomising the memory address the stack, heap, shared libraries (e.g., libc), and program image (e.g., the text segment) are loaded [7].

The Linux kernel randomizes the stack, heap, and shared libraries, but not the program image. Programs may, however, be compiled into Position Independent Executables (PIEs), which can be executed regardless of the absolute address the program image is loaded at. In practice, this is only done for a selected group of programs, because doing so introduces a performance overhead at runtime [7].

Windows Vista and 7 can randomize the locations of the program image, stack, heap, and libraries, but only if the program and all of its libraries opt-in to ASLR. If they do not, the process memory image is left unrandomized [7]. In Windows 8, ASLR can be forced on even for images which are not ASLR-aware [4].

Discovering input processing problems

A system can be viewed as having intended functionality, triggered by valid input, and unintended functionality, triggered by unexpected, typically invalid input. During software testing, unintended functionality is often neglected, as it is unspecified and the

10 **https://github.com/JonathanSalwan/ROPgadget**
triggers are often unclear. In defensive systems engineering, all input that cause undefined systematic responses are defects [9].

Typical input processing weaknesses include:

**Buffer Overflows**\(^1\) Copy operations with inadequate bounds checking lead to the write operations beyond the bounds of the target buffer. The affected buffer can be allocated on the stack or the heap. While only stack-based buffer overflow vulnerabilities are discussed above, heap-based buffers overflows\(^2\) can also cause vulnerabilities.

**Out-Of-Bounds Read**\(^3\) Read operations past the end, or before the beginning, of the intended buffer. This typically occurs when the pointer or its index is incremented or decremented to a position beyond the bounds of the buffer or the results of flawed pointer arithmetic end up referencing a position outside of the valid memory location.

**Uncontrolled Format Strings**\(^4\) Externally controlled format specifiers in the `printf()` family of functions can lead to buffer overflows and data representation problems, Format string attacks can potentially allow an attacker to read the contents or variables or even overwrite variables in program memory.

**Use-After-Free**\(^5\) Dereferencing memory after it has been freed.

**Reliance on Untrusted Inputs in a Security Decision**\(^6\) Software developers may assume that inputs such as cookies, environment variables, and hidden form fields cannot be modified. In reality, however, an untrusted actor in a way that bypasses protection mechanisms. The system might for instance execute SQL statements, the parameters of which have been influenced by a user.

**Uncontrolled Resource Consumption ('Resource Exhaustion')**\(^7\) A series of inputs might lead to Denial Of Service (DOS) conditions, such as with the case of the TCP SYN attack [3].

**Fuzzing**

Apart from static analysis, software flaws related to input processing can be detected via flaw injection [9]; the program under test is fed incorrect inputs, and its behaviour is observed. In contrast to static analysis, flaw injection is a form of dynamic testing.

---

\(^1\)CWE-121: Stack-based Buffer Overflow [http://cwe.mitre.org/data/definitions/121.html](http://cwe.mitre.org/data/definitions/121.html)

\(^2\)CWE-122: Heap-based Buffer Overflow [http://cwe.mitre.org/data/definitions/122.html](http://cwe.mitre.org/data/definitions/122.html)

\(^3\)CWE-125: Out-of-bounds Read [http://cwe.mitre.org/data/definitions/125.html](http://cwe.mitre.org/data/definitions/125.html)


\(^7\)CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion') [http://cwe.mitre.org/data/definitions/400.html](http://cwe.mitre.org/data/definitions/400.html)
During observation, the target program can be run under a debugger or dynamic test harness to collect diagnostics data.

With regards to software security, the objective is to ensure that a program fails gracefully when it cannot successfully do what it is supposed to do. Fault injection with the objective to uncover input processing issues is called robustness testing. Fuzz testing or fuzzing is a specific subclass of robustness testing. While fuzz testing can be something as simple as scratching the contacts of a hardware bus with a paper clip to induce chaotic signals, there are a number systematic of approaches to fuzz testing; generation-based or model-based fuzzing; mutation-based or sample-based fuzzing; and blind or random fuzzing.

**Generation-based or Model-based Fuzzing** In model-based fuzzing, test cases are generated based of a known grammar, typically a specification in Backus-Naur Form (BNF), Abstract Syntax Notation (ASN.1) or an Extensible Markup Language (XML) doctype. Starting from, according to the grammar, valid inputs one or more input fields or syntax tokens are manipulated when generating test cases. Due to knowledge of the input grammar, model-based fuzzing can guarantee a large coverage of the input space with a small number of test cases (in the order of hundreds to thousands of samples), but requires extensive preparatory work.

An example of model-based fuzzing tools is PROTOS\(^\text{18}\) test-suite developed at the University of Oulu, now commercialized by the privately-owned Codenomicon company \(^\text{19}\).

**Mutation-based or Sample-based Fuzzing** In sample-based fuzzing test cases are generated from a number of valid input samples by mutating them in various ways. Possible mutations include bit flipping, truncation, repetition, shuffling, token replacement etc. While sample-based fuzzing does not guarantee total input page coverage, it has the advantage of only requiring a number input samples, as opposed to a formal grammar as in the case of model-based fuzzing. Sample-based fuzzing also lends itself to automation, which is a good thing, since sample based fuzzing might require several order of magnitude larger set of test cases compares to model-based fuzzing, i.e. tens of thousand of test cases.

An example of sample-based fuzzers include Radamsa\(^\text{20}\), also developed at the University of Oulu.

A key factor in the effectiveness of sample-based fuzzing is the choice of input samples used to generate test cases. If the input is for instance compromised of files utilizing compression, mutating the compressed files themselves will most likely produce invalid input files which only exercise the decompression logic of the target program.

The process of reducing a large set of samples into a small set, while maximizing the extent to which the samples exercise the system under test is called corpus distillation \(^\text{9}\). An general overview of the process is shown in Figure 5. One way to accomplish this

\(^{18}\)https://www.ee.oulu.fi/research/ouspg/Protos

\(^{19}\)http://www.codenomicon.com/

\(^{20}\)https://www.ee.oulu.fi/research/ouspg/Radamsa
is to obtain a large number of valid input samples (in the order of tens of thousands); instrument the target system in order to perform code coverage analysis; and run each sample input through the instrumented system. With the results of the coverage analysis, the samples which provide the most variation in the coverage results can be selected with the assumption that, when fuzzed, test cases generated from the particular set of samples has the highest probability of exercising code paths deep within the program logic, where previously unknown defects might be found. The number of samples selected based in the analysis can be several orders of magnitude smaller than the original input set. A key property of corpus distillation is that it might not be necessary to perform the analysis on the actual system under test, but with another, similar, system, i.e. one that processes similar data.

Useful code coverage metrics for corpus distillation include basic block vector analysis and function call graphs. Basic block vector analysis revolves around determining unique blocks of instructions between branch instructions. Different sets of basic blocks correspond to different execution flows. Basic block vectors for programs can be generated with valgrind\textsuperscript{21} (see Listing 1). Function call graphs operate on a higher level of abstraction, but can still provide useful information when differences between distinct program runs are compared.

\begin{verbatim}
valgrind --tool=exp-bbv --bb-out-file=bbout --interval-size=10000 a.out
\end{verbatim}

Listing 1: Basic block vector generation with valgrind

Blind or Random Fuzzing In random fuzzing, a seed input sequence is transformed at random, without no or very little knowledge of the internal structure of the input. Usually, random fuzzing is the least effective approach and typically requires the largest number or test cases (hundreds of thousands or more) in order to produce results, but has the advantage of requiring very little preparation and being the easiest approach to deploy. It can also be the only possible approach in cases where unique or one-off inputs cannot be replayed to the system under test.

An example of a random file fuzzer is the MiniFuzz File Fuzzer\textsuperscript{22}, developed at Microsoft.

\textsuperscript{21}http://valgrind.org/
\textsuperscript{22}http://www.microsoft.com/en-gb/download/details.aspx?id=21769
The result of observing a system being fuzz-tested is usually dropped or rejected input (if all is well); a crash; or a hang. It can be argued that fuzzing has an advantage compared to static analysis in that fuzzing does not produce false positives; a crash is always indicative of a software defect when systems are engineered defensively. However, not all crashes are indicative of software vulnerabilities. Once a test case leading to unintended behavior has been identified, diagnostics information obtained from the test harness can be used to perform exploitability analysis of the crash, in order to prioritize defects which are directly related to security issues. This activity is known as triage, similarly to the medical term describing determining the priority of patients’ treatment based on the severity of their condition. A testing harness can be build using tools such as valgrind or AddressSanitizer\textsuperscript{23}, or full fledged binary instrumentation frameworks such as PIN\textsuperscript{24}.

Exploitability analysis can to some extent be automated based on a set of heuristics, such as if the crash occurred as a result of a return instruction; if the crash occurred as a result of use-after-free error; or if the crash occurred during a branch instruction [9]. Automated exploitability analysis is only indicative; a defect categorized as non-exploitable can indeed be useful in an attack, even if it's not the primary vulnerability. Some automated exploitability analysis tools can calculate a hash of the call stack in order to detect crashes that occur as a result of identical call chains. This can be useful for excluding duplicate results. Examples of exploitability tools include !exploitable\textsuperscript{25} for WinDbg, CERT Triage tools\textsuperscript{26} for gdb on Linux and CrashWrangler\textsuperscript{27} for Mac OSX.

**Conclusion**

Native code programs can exhibit multiple weaknesses with potential security ramifications. Attacks such as buffer overflow exploits can be devastating to such a degree, that a number of compiler and operating system-level safeguards, such as stack canaries, W\&X and ASLR, have been widely adopted to mitigate these. However, although support for such mechanism have become widespread, they are not always enabled due to compatibility and performance reasons. Furthermore, these mechanisms are merely a last line of defence. Proactive measures to be taken during the software development process to avoid vulnerabilities caused by software bugs include code review using static analysis tools and robustness testing using a fuzzer. In the context of defensive systems engineering, fuzzing has the advantage of not producing false positives. In fact, fuzzing often identifies so many crash conditions, that these have to triaged according to their exploitability in order to address the ones which are most likely to cause software vulnerabilities first. Exploitability analysis can to some extent be automated using various tools.

\textsuperscript{23}http://code.google.com/p/address-sanitizer/
\textsuperscript{24}http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
\textsuperscript{25}https://msecdbg.codeplex.com/
\textsuperscript{26}https://www.cert.org/vulnerability-analysis/tools/triage.cfm
\textsuperscript{27}https://developer.apple.com/library/mac/technotes/tn2334/_index.html
References


