Lecture 1 – How Software Breaks

Summary: This Lecture Started by introducing a number of the key terms and concepts that will be used throughout the course, such as vulnerability, exploit and attacker. An example of a specific exploitation was given via an explanation of a buffer overflow attack, which lead on to the software testing technique of 'fuzzing' or 'fuzz testing' which is used to examine software for such vulnerabilities.

Terminology

Although (as with many domains, especially emerging or rapidly developing ones) there are varying, and at times conflicting, usage of terms, here we will outline the terms and their meanings to be employed in this course.

Software 'breaks' when it acts in a way that was not originally intended. Although a program or system may be operating fine, and successfully performing all the tasks it was designed to do, it still has the potential to fail. Those possibilities for failure have the potential to stop the system doing its job (crashing it or otherwise disrupt it), or to be manipulated for malicious intent (to steal or miss-use the system).

When considering software security, a weakness is any type of flaw, fault or bug that has the potential to cause the system to fail, or 'break'. Software weaknesses have a classification system, the Common Weakness Enumeration (CWE)\(^1\), which aims to catalog and record weaknesses to support secure software development. A vulnerability is a single specific security problem which has the potential to be exploited in an attack. Vulnerabilities are similarly classified using Common Vulnerabilities and Exposures (CVE) codes\(^2\). As indicated, an exploit is the act of using one or more vulnerabilities in order to cause unintended results in the system.

Vulnerabilities can take the form of bugs or flaws. A bug is a software fault, and as such is an implementation issue. In contrast a flaw is a logic or design issue. An example of a bug would be a plug-in which has faulty code leading to a vulnerability. A design level flaw would be the use of that vulnerable plug-in in a larger program introducing that vulnerability to the whole system.

The attack surface of a system is everything that an attacker has access to through which they can interact with the system. Attack vectors are the specific routes an attacker can employ to access the system and exploit a vulnerability. The attack surface is the sum of all the attack vectors, including input fields, protocols, interfaces and services.

An attacker is someone who performs actions against a system in an effort to discover and exploit vulnerabilities, however for the purposes of this course we do not assume an attacker has malicious intent. The term attacker encompasses those individuals who are 'attacking' the system for the benign purposes of testing and securing it against malicious attacks. The term threat, or threat actor is used to refer to attackers who are likely to have malicious intent.

---

1 Common Weakness Enumeration - https://cwe.mitre.org/
Interestingly enough, in *Secure coding: Principles and practices* Graff & Van use the term 'attack' to refer to 'any maliciously intended act against a system' (page 6) regardless of goals or objectives. They go on to describe the components of an attack as follows:

- **Goals** – the overall objective of the attack, such as crashing the system, damaging th system or stealing money or data.
- **Sub-goals** – smaller steps which are an integral part of achieving the main goal, such as gaining unauthorised access to the system.
- **Activities** – The actions undertaken to achieve the sub-goals, such as stealing login credentials or masquerading as a trusted device.
- **Events** – The results of the activities, such as improper access being granted, storage space being exhausted or the process crashing.
- **Consequences** – The physical results experienced by the business as a consequence of the events. This manifests itself as missing money, corrupted data or loss of use of the system for normal business activities.
- **Impacts** – the overall effect on the business, such as loss of revenue, reputation or partner/consumer trust.

When differentiating between the term 'attacker' and 'threat actor' as described above, it is possible draw a line in the above sequence between the **event** and **consequences** phases. An attacker such as an in house software tester or an external security consultant would endeavour to breach the system by achieving the events of gaining access or crashing the system as described above. This would allow them to identify potential threats to the system to develop safeguards against them, however the process would stop short of having any negative impact on the company (consequences and resulting impacts). Attackers with malicious intent – ‘threat actors’ – however would be attempting to achieve their goals of stealing, damaging or disrupting which will result in the negative consequences and impacts for the business.

**Buffer Overflow Attack**

Vulnerabilities in software relate to three broad categories depending on where the weakness was introduced during the development process: Architectural/Design – while the application is being planned / designed, Implementation – whilst the code is being written and Operational – after the application is running.

As an overview of how different types of attack relate to these categories we will list a number of attacks by level:

- **Architectural/Design level attacks** – Man-in-the-middle attacks, Race conditions, Replay attacks, Sniffer attacks, Session hijacking and Session killing attacks.
- **Implementation level attacks** – Buffer overflows, Back door attacks and Parsing errors.
- **Operations level attacks** – Denial-of-service attacks, Default account attacks and Password cracking.

---

As an example of a specific attack we investigated the buffer overflow attack. To relate this to our earlier terminology, buffer overflow attacks use the general weakness of getting the attackers code to run on the target system for their own purposes and as such are an exploit of the specific buffer overflow vulnerability.

A buffer overflow is when data written to a buffer extends beyond the predefined bounds of a memory buffer and overwrites (and consequently corrupts) neighbouring memory addresses⁴. A buffer overflow attack exploits the weakness of being able to manipulate memory addresses commonly by changing memory addresses near the buffer to point to malicious code injected by the attacker.

Buffer overflow attacks are a binary exploitation where the binary code of the system is manipulated. This type of vulnerability is generally introduced to the program during development (where potentially dangerous user input is not proactively managed to prevent potential buffer overflow conditions).

Buffer overflow attacks rely on a manipulation of the stack region of a programs memory space.

---

Figure 1. Memory structure showing the stack and the stack frame structure⁵

A programs executable code is loaded into the text section in low memory, with the stack in high memory being used dynamically during run time. The stack grows downwards by adding stack frames when temporary memory space is required (figure 1)

---


⁵ From lecture slides by Antti Vähä-Sipilä - https://wiki.helsinki.fi/download/attachments/127960693/Session1.pdf?version=1&modificationDate=1394552259751&api=v2
When a function call is made a new stack frame is placed on the stack containing the call parameters, the return address to the next instruction in the text section (which allows for the program to continue executing after the function has finished) and an allocated space for local variables (figure 2a). The local variable section holds all the variables required for this function call, each having their own buffer.

Each variables buffer size is predefined and fixed, for instance 16 bytes to hold an array of 16 characters (or more precisely, if considering a null-terminated string, 15 characters plus a string termination null character).

Figure 2b shows a simplified stack frame with the return address (pointing to the next executable line of code in the text section) and an empty buffer for a single variable of 16 bytes, with figure 2c showing a valid use of the buffer holding a string of less than 15 characters – 'Hello World' in this case.

![Stack frame structure](image)

**Figure 2.** Stack frame structure (a), Stack frame with empty buffer for a single variable (b) and with a valid string (c).

If the input data is not checked, it is a possibility that if too much data is injected to the buffer it may start overwriting memory beyond the buffer. Figure 3a shows the results of inputting 20 'A's and 4 'B's – AAAAAAAAAAAAAAAAAAAAAAAABB

BBB', where we can see the Saved Frame Pointer and the Return Address being overwritten with the input string.
To turn this buffer overflow into an attack the attacker aims to overwrite the Return Address with a memory address that points at his own malicious code (figure 3b) so that the processor thinks that this is the next instruction and executes it.

One common tactic employed in stack buffer overflow attacks is the NOP-sled (figure 3c). An NOP-sled is where a number of no-op machine instructions are strung together in a block to create a larger target for the redirected jump address. This solves the problem of not knowing exactly the position of the buffer and so not knowing exactly the address of the malicious code (as in figure 3b). If the attacker can estimate the return address to land anywhere within the NOP-sled then execution will just 'slide' down (or 'up' with layout of figure 3) the consecutive no-op's until the malicious code is hit and executed.

In practice the majority of buffer overflow vulnerabilities result from C or C++ code and are mostly linked to functions that do not specify the size of the destination array, such as strcpy(), strcat(), and sprintf(). CERT report that buffer overflow (classified as Violation of Memory Bounds vulnerabilities) are “the single largest, and today probably the most damaging, class of vulnerability in the CERT/CC database.”

---

Fuzz testing

Inputs to a system are implicitly ways to interact with the system and therefore constitute (a major?) part of their attack surface. Even with the best defensive development and static code analysis all software has the potential to fail under unexpected (or new from the time of development) conditions. One approach to examine systems for unknown bugs (which have the potential to be security vulnerabilities) is the process of fuzzing.

Fuzz testing, or 'Fuzzing', is a security testing technique used to examine software for flaws and potential security vulnerabilities. It has become popular for being a quick and cost effective method of testing large applications for serious security defects.

When looking at a program you expect it to function as intended when supplied with a valid form of input, such as a PDF viewer correctly displaying a document on screen when asked to open a valid PDF file. If that program is supplied an unexpected or invalid input we would like to see it handle the anomaly gracefully, tell us it's the wrong file type (or corrupt), and quietly sit there waiting for a meaningful PDF to open. This however does not always happen and invalid inputs can trigger unintended behaviour such as crashing.

Unintended behaviour in a program, from a security perspective, can indicate a vulnerability which could potentially be exploited. Fuzzing tests programs for flaws by sending malformed, or semi-malformed data into that program and seeing if it crashes, and is know as 'flaw injection'.

Fuzz testing techniques fall under 3 main categories - Random, Model-based and Model-inferred-based fuzzing. Random fuzzing randomly manipulates bits within the source file (such as bit swapping), which is easy to implement and can be useful, but may not be as powerful as a targeted approach. Model-based fuzzers use a detailed knowledge of the syntax of the target to produce test cases tailored to the grammar elements of the target code, but they require a large investment of effort to generate a suite of tests. Model-inferred-based, or sample-based, fuzzing uses examples of the programs input to produce fuzzed test cases. This approach, although not as detailed as model-based tests, is more focused than just random data; and is quick and easy to start testing using tools such as Radamsa.

At a basic level the process of Model-Inferred-Based fuzz testing takes valid data entry cases for a program, such as JPEG files for an image viewer, alters them slightly by passing them through a fuzzer to create invalid cases which are then injected into the target program (figure 4). The idea is to break the target program by causing an unexpected exit from the program or crash.

---

Radamsa\textsuperscript{10} is an automated black-box fuzzing tool which operates as a Model-Inferred-Based test case generator. It is 'black-box' as it has no knowledge of the targets internal structure, but it takes a sample of valid inputs and applies a number of fuzzing techniques to generate a wide variety of malformed input test documents.

The Radamsa tool is actually a collection of multiple distinct fuzzer algorithms performing alterations from simple bit swapping, through repeating part of the data, to cutting segments from one sample file and inserting them in another. These algorithms run in parallel to create a large volume of test cases to run against the target application.

When fuzzing is applied on a large scale an extra phase of 'Corpus Distillation' is undertaken to improve the breadth of representation of the sample files\textsuperscript{11}. Corpus Distillation takes a large number of valid files for the target application which are then analysed to calculate a smaller set of files which best represent as wide a number of code paths as possible. These files are then fuzzed to create a very large number of test inputs for the system\textsuperscript{12} (figure 5).

As an example, Google Security Team reported using a corpus distillation approach to fuzz testing Adobe's Flash Player\textsuperscript{13}. They analyse 20 terabytes of SWF files to reduce them to around 20,000 files (taking a week on 2000 CPU cores). Those 20,000 files were used as a seed base to generate a huge (unspecified) number of test cases (taking 3 weeks to generate on 2000 cores). This operation identified about 400 unique crash cases which related to 106 individual security bugs.

\textsuperscript{10} Radamsa by the Oulu University Secure Programming Group - http://code.google.com/p/ouspg/
Once the fuzzed (malformed) input files have been generated the process of injecting the test data to an application not only needs automating, but also the results of any anomalous results need capturing and recording. Dynamic analysis tools such as Valgrind\textsuperscript{14} can be configured to create such a testbed providing the potential to garner results from hundreds of thousands of test inputs.

Whilst most (hopefully) of the flawed inputs are handled gracefully and dropped by the application, any that create crashes or hangs indicate a software defect, as one benefit of fuzzing is that it creates no false positives – every crash IS indication of a bug. However, not every bug IS a security vulnerability, so once bugs have been identified they need assessing to prioritise them in order of risk to the system, a process known as triage.

**Conclusion**

Software breaks. It just does. How much, how badly and under what conditions vary greatly. These failures, from a security perspective, constitute the weaknesses and vulnerabilities which potentially open the system up to be compromised by attackers.

Even in this first lecture of the course, it has become clear the importance of IT professionals embracing this perspective and striving to develop software informed by better knowledge both of known vulnerabilities and the methods of mitigating their effects.

In addition this perspective acknowledges it is not possible to produce a 'perfectly secure' piece of software (even with the best defensive development and static code analysis) and so testing, such as the fuzzing described above, constitutes an essential and integral part of making systems as secure as is possible.

It seems, to this student, that improvements to software security will come not only from improved education of practical security issues and best practices, but also from a shift in the mindset of developers about the fundamental nature of secure software development due to that knowledge.

\textsuperscript{14} Valgrind: http://www.valgrind.org/