How Software breaks

Prefaces
Software security relates to designing and developing software such that they cannot break in case of any intentional or accidental attack. Security is primary concern for software development projects since their inception. Breaking a software does not necessarily equates “hacking” it, rather it is a phenomenon of exploiting any vulnerabilities in design or implementation of software. An attacker can exploit there vulnerabilities to break into a software and compromises its functionality or gain unauthorized access to the system. To develop better understanding of how a system breaks and what can be possible ways of exploring and exploiting any vulnerabilities in a system, first we need to understand some common terminologies.

Secure design
A product is said to have secure design if the whole program is designed to be secure and it resists any attempts from an attacker intend to break the program.

Weakness, Common Weakness Enumeration
A software weakness is some implementation bug or design flaw which may be left unaddressed at the implementation time and can be used later for exploiting the software. MITRE Corporation provides free access to an online repository “Common Weakness Enumeration (CWE)” which contains weaknesses found generally in software products [1, 2]. CWE facilitates the use of effective tools to identify and resolve different software weaknesses which may arise in a system. CWE is maintained by group of experts from academia, industry and government agencies, in order to maintain the diversity of content available to the user. Large enterprises use this repository to compare and evaluate different products before using them in production setup. CWE has three levels or tiers for classification of weaknesses.

- Top most tier distributes the weaknesses into few large groups among people from management, academia and research.
- Second tier provides the definitions categorized by experts, system administrators and developers.
- Third tier gives the proper definitions of these weaknesses to the common users.

The entries in CWE are assigned with an identifier which can replace the proper description of any weakness in a security report. For example weakness ID 425 refers to forced browsing or direct request weakness [3]. In this weakness, a web application doesn’t include proper authorization to restricted URL, scripts etc. This weakness can arise at design time, implementation time or
operation time and is present in all programming languages. Common consequences of this weakness includes access to confidential application data e.g. query string manipulation.

**Vulnerability, Exploit Common Vulnerability and Exploits**

The threats to software systems can be categorized into exposure and vulnerability. Vulnerability is a factual weakness about a computer, server or network which definitely has some risk associated whereas exposure is a situation considered as risk by a few people but not everybody [4]. A software vulnerability is a software weakness which can be used by an attacker to reduce system integrity [5]. An attacker can exploit these vulnerabilities to break into a system and compromise system functionality. A vulnerability is a security risk when it is related to high value content and not every vulnerability is necessarily a risk e.g. if query string manipulation enables a user to log into web application as administrator, it is high risk vulnerability but if only allows the user to traverse upward in html file directory (which doesn’t contain important/ confidential data), it is negligible. Like CWE, “Common Vulnerability and Exposure” (CVE) is developed for reporting common vulnerabilities found in software systems [6]. In CVE, every vulnerability initially is an exposure unless it is universally considered as vulnerability from all the aspects. In CVE, every vulnerability has an associated identifier which includes the year it was found and the order it was added to the repository list. For example CVE-2012-2001 gives information about the security vulnerability in HP SNMP agents for Linux which may result in cross site scripting and URL redirection [7]. The identifier shows that this vulnerability was reported in year 2012 and the sequence number was 2001.

1. Those which result in direct compromise e.g. a vulnerability exploited in the security mechanism of operating system. An operating system bug is a vulnerability.
2. Those which are dangerous in the long term e.g. high speed internet connection pose no serious threat to the system in immediate future but it is much easier to hack into a computer with high bandwidth as compared to one with dial up modem connectivity. A high speed internet is considered as exposure.

Some of the possible vulnerabilities are not publicized and they are used as backdoors by some actors, mainly spy, government agencies [8]. These vulnerabilities may be specially introduced in software during design phases for specific purposes. There are two common types of vulnerabilities.

An exploit is when attacker uses a software vulnerability to get into a software system and for any malicious purpose. An exploit can be a piece of code or data which is injected to system through some system vulnerability and result in unintended behavior in software or hardware[4]. In common scenarios, an exploit is meant to gain unauthorized control over some device or privilege escalation etc.

An attack is referred to any activity that tends to compromise program execution to take unauthorized control of program or system. Internet engineering task Force [9] defined an attack as

1. Those which result in direct compromise e.g. a vulnerability exploited in the security mechanism of operating system. An operating system bug is a vulnerability.
2. Those which are dangerous in the long term e.g. high speed internet connection pose no serious threat to the system in immediate future but it is much easier to hack into a computer with high bandwidth as compared to one with dial up modem connectivity. A high speed internet is considered as exposure.

Some of the possible vulnerabilities are not publicized and they are used as backdoors by some actors, mainly spy, government agencies [8]. These vulnerabilities may be specially introduced in software during design phases for specific purposes. There are two common types of vulnerabilities.

An exploit is when attacker uses a software vulnerability to get into a software system and for any malicious purpose. An exploit can be a piece of code or data which is injected to system through some system vulnerability and result in unintended behavior in software or hardware[4]. In common scenarios, an exploit is meant to gain unauthorized control over some device or privilege escalation etc.

An attack is referred to any activity that tends to compromise program execution to take unauthorized control of program or system. Internet engineering task Force [9] defined an attack as
An assault on system security that drives from an intelligence threat i.e. an intelligent act that is deliberate attempt (especially in the sense of a method or technique) to evade security services and violate the security policy of a system.

Attack can be of active or passive [10] where

- **Active attack** affects the system program and resources by directly interacting with execution process and system resources.
- **Passive attack** steals the information from the system without affecting any functionality.

An attack can be launched from **inside** the security parameters of attacked entity e.g. a disgruntled employee defacing company servers to avenge the disrespect he received. It can also be attacked from **outside** the security premises by some attacker who might want to take control of these servers for any possible reason. Mainly the attacks launched from inside the security parameter are very rare but also extremely lethal since the attacker have to bypass all the firewall and access parameters and he also has quite an insight of system how it works. He can make a better idea for crafting attack. In worst cases, he might have physical access to the system resulting in serious threat to security.


```
Any kind of malicious activity that attempts to collect, disrupt, deny, degrade, or destroy information system resources or the information itself.
```

Based on the nature of attack and vulnerabilities they target, they can be classified into three major categories.

**Architecture level attacks**

These attacks are most hard to find but once found can be very lethal for the system [12]. Design or architecture level attacks are not always software design flaws, but it may be due to reliance on some functionality provided by another entity which gets compromised e.g. if a plugin in WordPress gets hacked and affects your blog which is using that plugin, it is not necessarily your fault to use that plugin but of word press whose platform feature ended as security threat to all the applications developed over it.

- **Man in the middle attacks** first steals the information flowing between two communicating bodies and then masquerades itself as one of the parties. This kind of attack can be countered by using encrypted communication between two bodies.
- **Race condition attack** is an attack in which attacker uses the slight window between execution of instructions in operating system. The attacker repeatedly use these windows of opportunities to execute his own codes at same level of privilege. To mitigate this attack, non-atomic operations should be used to avoid any windows during execution.
- **Replay attacks** are similar to man in the middle attack. For replay attacks, attacker intercepts communication between two entities and then replay the conversation for some
ill purposes. Impersonation can be very serious and it should be avoided using encrypted communication.

- **Sniffer attack** also lets an attacker sniff through the traffic going on between two bodies. It can be avoided using switched networks and encrypting the network traffic.
- **Session hijacking** mechanism exploits TCP/IP stack to hijack the established connection between two bodies (server and client) and use this connection as one of the participant entity. This attack can be subverted by using cookies and encryption.

**Implementation level**

These attacks are possible due to issues during the implementation of product, when developer dint pay attention to different miner issues [12]. These attacks are most common in routine. These attacks include

- Very common **buffer overflow attack** which occur when user input overruns the buffer space allotted to it. It can be avoided by reading sub-string into fixed length buffers.
- **Back door attacks** are implemented by the developer to keep a backdoor into program right at the time of implementation so that anyone who knows about it can enter the system.
- **Parsing error attacks** occur when user input is not properly parsed before processing it. A fairly crafted input string can let the user access the resource which are not supposed to be accessed by him. This issue can be mitigated by getting the code analyzed by some expert, and processing the input characters to make sure they are safe input.

**Operational level**

These attacks are faced in the production environment [12] such as

- **Denial of services attack** in which host system is overwhelmed with unusable requests for services and legitimate users cannot receive services anymore. It can be resolved by carefully planning the resource allocation at the time of deployment as well as effective load sharing to provide uninterrupted services.
- **Default accounts** attack tend to use any possible preconfigured passwords for program or operating systems. A solution for this attack will be to remove all default accounts and persuade users to use strong passwords.
- In **password cracking**, an attacker may try to crack the password of legitimate user by performing guesses to his password. These activities are done by dictionary or brute force attack in general. The program should be able to identify any such attempts and block them before they can succeed.

**Attacker**

Attacker is any actor who exploits a vulnerability of a system. An attacker doesn’t necessarily intend to destroy or compromise a system but he can be a security professional doing penetration testing and security evaluation of any system. There can be two types of attackers to a system

- **Black hats**: those who exploit vulnerabilities to gain ill-legal advantages.
- **White hats**: also known as ethical hackers are security professionals who evaluate system functionality by finding possible vulnerabilities and exploits.
Attack vector and Attack surface
Any resource which an attacker can get use to gain entry into system and (may) transport data inbound or outbound to the system is known as attack vector [13]. In software products, common attack vectors may be interfaces, protocols, ports and services etc. The collection of all attack vectors is known to be attack surface [14]. Commonly, an attack surface is a state of system which may accept ill-formatted inputs and result in putting system in a compromised state.

Controlling the attack surface restricts the foreign actors into the system but it doesn’t do anything in terms of securing the system in case of an attack. It only provides an entry barrier for any attacker accessing the system, who once entered can cause serious damage to the system. Attack surface analyzed is a tool used and provided by Microsoft for analyzing and track the changes made to the attack surfaces of a system.

Threat & Risk
A threat is an actor who may exploit a vulnerability identified in the system and risk is the estimate of likelihood that a threat will exploit that vulnerability [15].

A threat to a system can be any person or organization who want to break the system or disturb its functionality to maximum possible. It can be a natural disaster resulting in system outage. It can be any circumstance or capability which takes system to a state where it becomes vulnerable or it can be accidental in case of computer malfunctioning or underlying support application breakage.

Threat agents
Threat actors or threat agents can be any single or group of factors which can result in breaking a system [15, 16]. In practice, anything can be threat agent including intelligent user, ignorant system user, and complex security measures to a system or a hacker.

Bugs and Flaws
A security bug or flaw is design or implementation time anomaly which can later result in system exploitation. Any anomaly in system will result in the advantage of an attacker for breaking into or compromising system or software integrity. Bugs and flaws arise in a software due to design and implementation time issues.

How adding more security feature could can make system less secure?
A common estimate from a person may be that improving security measures for a system will lower the attack surface of the system but this is not always the case. One has to keep balance between the level of security and attack surfaces since increased security possibly increase the attack surface [17]. There can be several types of interaction and each one of them has different depths, hence the cost and duration of security testing becomes huge. To overcome time and cost barrier, developers feel “satisfied” i.e. they oversee different aspects of system with a thought that it never had a problem or will never be considered by the user or attacker. Also, when there is not enough resource to conduct proper inspection, developers usually hide the things and hiding things can later prove to be a problem.
Consider a scenario where developer encrypts all the network traffic and illegitimate traffic flows through the network in encrypted form without coming into notice of traffic monitors etc. Similarly, if the whole hard drive is encrypted, there may be a possibility that malware also sits in this hard drive in encrypted form. Keeping a strong password never ensures the safety and integrity of data, it just blocks the access to data. The fact that data stored using strong passwords can be illegal or malicious data which is overseen by satisfied developer.

Adding more security features to the system will need installing new add-ons and applications which means more attack vectors. Whenever a system is made secure, a balance between security and functionality should be kept in perspective i.e. a system cannot be made secure by considering its current state only, future updates and maintenance should be considered while applying the security features. Updates are also a serious issue which should be considered because in case of updates e.g. java updates, a third party i.e. software updater can alter our system security controls without our knowledge [17].

Common types of exploitation

Binary exploitation

Binary exploitation includes different techniques to hack into a system or software for executing malicious codes, privilege escalation, and access to sensitive resources on the target machine. These exploitations may occur [18]due to

- Vulnerabilities in the programming approach during the implementation.
- Improper testing and verification of final product against different common attacking approaches.
- Out of date software, support libraries, OS support etc.
- Any premature, prototype piece of code included in the final product without conducting proper testing.
- Use of module or functionality which was not designed to use in high risk environment.

First step of binary exploitation is to find a way to insert a malicious code into the system which can later open different inlets to the system [16]. With administrator privileges, a malware can do unexpected damage to the system. Forced buffer overrun in one of the very basic techniques to break into a software and execute the malicious piece of code in legitimate environment.

Buffer over flow attack

Buffer over flow attacks are one of the very first and well known exploit written for computer programs. Buffer overflow attack exploits common weakness where a program fails to set a bound while writing user input to the memory such that input overwrites the adjacent memory blocks. An attacker can use this attack to write his own code in the memory and use address pointers to execute his code during program execution [19]. Program memory contains stack and heap where the former grows downwards and later grows upward in the memory as shown in Figure 1. Buffer overflow attacks can use both stack and heap but stack based buffer overflow attacks are more common.
1. **Stack based buffer overflow**

When a process is called, a stack frame is incepted and placed on stack with its call parameters, the return address to text section and input variable buffer space where text section stores the address of next instruction. Since buffer size is fixed, if the user input exceeds this buffer size, it overwrites the adjacent memory area which contains pointer to text section. An attacker may change the return address (from text section) to arbitrary address e.g. memory address of malicious code to execute his own code. A horizontal view of input buffer is shown as under to see how it gets over written. In first phase, input variable space is empty.

<table>
<thead>
<tr>
<th>Input variable space</th>
<th>Variable</th>
<th>Return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00 00 00</td>
<td>AB 0b 01 03</td>
<td>12 11 1A BB</td>
</tr>
</tbody>
</table>

When user input is written to the memory area, it grows but most of the part is still empty.

<table>
<thead>
<tr>
<th>Input variable space</th>
<th>Variable</th>
<th>Return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA 01 04 15 00 00 00 00</td>
<td>AB 0b 01 03</td>
<td>12 11 1A BB</td>
</tr>
</tbody>
</table>

When user input exceeds a threshold it starts overwriting the other values stored in stack. Up to this stage stack overrun will only result in program corruption but if an attacker crafts overwritten data in such a way that he replaces the return address with the address pointing to the malicious code, the situation becomes dangerous.
The program first crashes due to stack overrun and then tries to maintain itself by going back to the return address which ideally contains address of next instruction. Since the return address is overwritten, it will return to malicious code placed by attacker and execution will compromise the system and program.

![Figure 2: Return call to stack overwritten by the attacker](image)

It is not highly unusual that attacker doesn’t know about the exact address where his overwritten code resides. In case attacker doesn’t know exactly what return address should be, he pads the malicious code with NOP instructions, on both sides. As a result, if return address falls in the range of padding, malicious code gets executed.

Majority of buffer overflows occur in applications containing native C code where functions e.g. `strcpy()` etc., which do not give memory size explicitly are used.

**NOP (No-op) instruction** does nothing but increments advances instruction pointer or stack pointer to next execution

2. **Heap based buffer overflow**

This attack occurs when the large amount of data is written over memory heap area allocated to a program. The attacker may flood the heap memory with his own data to overwrite the content of heap and virtual function table [20]. The overflowed data overwrites the headers of next heap areas in the memory and when heap management function is called and it accesses the overwritten
heap memory areas, the headers may raise an exception in heap management function which can be used by attacker. Changes in heap management function will demand change in attack routines that is why these attacks are difficult and therefore less common compared to stack buffer overflows.

*Use after free* heap attacks are different from heap overflow. These attacks require that heap entry is created and use it once again after HeapFree() is performed. The use of these free heaps can allow an attacker to control the execution process.

**Ways to prevent buffer overflow**

Stack buffer overflow techniques are very old and many schemes have been developed in order to mitigate these attacks [8]. These technique mainly work by

- Identifying that stack overflow has occurred and suspending the program execution.
- Preventing the execution without detecting stack buffer overflow.
- Randomizing the memory address so that attacker cannot relocate the malicious code in the memory.

**Stack canaries and stack cookies**

This technique works by some variable written over the stack to identify the integrity of stack. A small trigger is chosen at the start of program and written at random position in the stack [8] mostly before return address value. The *canary* value is written in such a way that it must change before return pointer in case of stack buffer overflow. When return pointer routine is called, canary value is checked and any value for stack canary other than the one written at start, will halt the program execution. This technique is very effective to control buffer overflow since the attacker has to think of some smart way to access the return address pointer.

**No execution in stack and heap area**

This feature is implemented as part of *Data Execution Prevention* (DEP) mechanism used as security feature in most of modern operating systems [21]. This technique prevents the execution of any code (other than executable instructions) in the memory region, whether stack or heap. DEP is useful in preventing buffer overflow exploits since it disallows the policy of *Write XOR Execute* i.e. you can either write to a memory location or you can execute it [8]. DEP can be hardware enforced or software enforced.

*Hardware enforced DEP* enables CPUs to mark the memory areas which can only hold data. These areas are marked as NX (Non eXecutable), XN (eXecute Never) and if any process asks for execution in these memory addresses, the execution is halted. In latest version of windows, DEP status can be viewed from processes tab in windows Task Manager [22].

Microsoft calls software DEP “*Safe Structured Exception Handling*”. It checks if the exception raised is registered in function table and program can be built with it. *SafeSEH* doesn’t use NX bit and is different from protection [22].
Request oriented programming (ROP)
To walk around the DEP security feature, an attacker can overwrite stack with return addresses pointing to short code snippets, commonly known as gadgets. Each code snippet performs some basic functionality and returns to next instruction pointed. An attacker thus chains these small snippets together for achieving some certain functionality without executing any code [8]. These calls work in effect of disabling memory execute protections and allow shell code to run as normal so the execution never vectors to the stack.

Return oriented programming is a variant of return to libc method in which an attacker can use return to libc method from shell code creation in which malicious payload will load stack with proper call stack so that execution is vectored to chain of standard library calls (e.g. system(), printf() etc.) [21].

Address space layout randomization
Memory address randomization technique is used to counter the ROP approach. This technique randomizes the memory address of the binaries loaded so that attacker cannot effectively chain the binaries together [21]. The technique is called as address space layout randomization. It can be implemented in multiple ways. E.g. windows executable are linked to linker at runtime so that they can run from different memory addresses whereas Linux compiles binaries (ASLR-enabled) into Position independent executable PIEs, for their execution from arbitrary locations. In some cases, dynamic libraries are prelinked to enhance the performance so libraries need to be randomized before prelinking. Prelinking can be done only once in the creation stages and it cannot be changed later. Hence PIEs on 32bit and x86 systems is not an effective idea.

ASLR implementations and issues
NX can be encountered using Return Oriented Programming in which attacker examines the target binary with gadgets, essentially short snippets of code and chain these gadgets together to obtain required result. Once the gadget chain is discovered, the program stack is rewritten by a series of return addresses to these snippets so that program executes in controlled manner resulting in desired output.

DEP is not useful to deal with old programs which do execution in memory areas. In case of such software, DEP should be disabled for smooth execution which makes the applications vulnerable which rely on DEP for security against exploits. The use of XN, NX, stack canaries is also not possible universally. In certain platforms, shared libraries with no NX support force their parent libraries libraries not to have NX. Stack canaries can also be missing in case they are not included in the code at compilation time, hence it can results in termination of legitimate programs without giving any error information [21]. This can be painful to diagnose. In case of ASLR, partial randomization can results in prediction of addresses of next binaries and ASLR necessarily alleviates its purpose. It may be possible that underlying kernel is not using ASLR and attacker can reach to software components running the protection mechanisms.
Fuzz testing
Fuzz testing, also known as Fuzzing is a simple yet useful technique to discover software program errors and security loopholes by inputting large number of random test cases based on valid inputs. The random data is known as fuzz [23]. Fuzz tester is a tool which indicates the potential causes of vulnerability identified using fuzz. Fuzzing was first developed in 1989 by Prof. Barton Miller’s research group at University of Wisconsin Madison. Fuzzing is primarily used in detecting security flaws in software systems.

Fuzzers are very useful to identify issues like buffer overflow, cross site scripting, invalid input cases etc. Many hackers use fuzzing to identify the set of possible vulnerabilities in the system which they later use to write their exploitation [8]. Fuzzers are not useful to identify software which do not directly interact with program to steal information e.g. spywares, viruses, Trojans and keyloggers.

Fuzz testing is very useful because it is simple and doesn’t require background knowledge of target e.g. black box fuzzing, as well as it is very cost effective.

Purpose
Fuzzing is mainly used for evaluating the software components by using set of anomalous inputs. It injects the system with bunch of scrambled inputs which are expected to trigger some unusual behavior of system. It is used to assess the quality of tests cases employed to find out any specific design or implementation issues [24]. It simulates the general world usage behavior for inputs to a system which is very useful for developer to test the functionality of a system in real world. In a test environment it is difficult to simulate the real world behavior due to lesser number of participants, this limitation is very well resolved by fuzzing approach.

Technique
In fuzzing, usual data accepted by any software e.g. images in case of fuzzing an image viewer, is used to develop fuzz which is then fed to the software for checking if any input triggers a bug in the program [25]. During development, programs are thoroughly tested to find if there is any security flaw in the design and implementation of the functionality but developers may not pay attention to many aspects, thinking that nothing can go wrong in here e.g. an image editing software also lets user to preview images in the hard drive without importing them to software. During analysis, the security and development team must have checked the image processing algorithms and their implementations working underneath to find any flaw or bug but nobody might consider the image preview functionality as an attack surface since it is only meant to preview and how can displaying an image may crash image viewer. There may be a case later when displaying some manipulated image result in breaking the whole software. Fuzz testing finds out such attack surfaces which are generally ignored by developers by considering them trivial. Fuzzing targets such scenarios by feeding random inputs to the program and see if any of them results in a program crash, halt or any other unexpected scenario. Fuzzers may use different technique to generate fuzz. Some of them are described as under.
Random fuzzing
In random fuzzing, completely random fuzz test cases are generated from a valid input set. The fuzz cases may be generated by simple bit flipping or replacements of input segments in the valid cases [24]. This kind of fuzzing is easy to implement but it may not necessarily be able to extract some interesting anomalies in software behavior since it relies on simple alterations in valid input [8]. Also, too many cases in random fuzzing may equate brute force testing scenario. This technique is mostly used by the intruders who do not have any knowledge about the system and inputs.

Model based fuzzing
Model based fuzzing is more intelligent approach than random fuzzing. It uses information i.e. grammar, about input to the system and based on that information it generate test cases [24]. Since this technique has background knowledge of input composition, the fuzzer may target pattern and areas in the input to build test cases which are more probable to hit the artifacts in the program. Although this approach is very useful for fuzz testing scenarios but it requires deep understanding of software functioning and how it handles the input. This thing can take a huge amount of time before actually fuzzing the system which might not be the case in penetration/ software testing scenarios [8].

Generation inference based fuzzing
Generation based fuzzing is more intelligent variant of random fuzzing. This technique works by taking large number of inputs for basis of developing fuzz inputs. All these inputs are analyzed to find specific patterns which may be interesting e.g. finding the mostly accessed memory area/ program code segment (it can be done by using tools e.g. VALGRIND etc). Fuzzer then takes small set of most interesting samples (probably ones hitting some code segment more often) from input set and generate very high number of fuzz test cases. These test cases are more likely to hit the vulnerabilities as compared to random fuzz cases. This approach is suitable for testing software when more time and resources are available for testing [8].

Corpus distillation
Corpus distillation is based on model inferred based fuzzing technique. In corpus distillation, large number of sample files are collected and they are analyzed to locate the frequently accessed areas by the files [26]. Then an algorithm is used to generate the files which achieve the code coverage of full set of accessed locations. This minimal set is very useful basis of fuzzing the program since it gives a decently large (manageable) number of files that exercise most of the unusual code paths. Corpus distillation enable to generate interesting new patterns from the legitimate inputs which can be very trigger some unexpected routines in program during processing. Figure 3 shows a flowchart of fuzzer which uses corpus distillation.
Types
There are three different types of fuzz testing techniques, listed as under.

White box testing
White box fuzz testing can also be known as smart fuzzing. White box fuzzing is used to find buffer overflows, unhandled exceptions, denial of service etc [27]. It requires good background efforts of tester to understand the system functionality e.g. by tracing runtime execution to find the program flow [24] and generate test cases which cover entire breadth of the system testing. It ensures that all the execution and targets path are hit during the fuzzing. White box testing looks for high ranked software quality testing and detection of all possible vulnerabilities in the system.

Grey box testing
Grey box testing falls between white and black box testing where tester may not have access to complete source code, function execution or anything but it tries to extract useful information from usual execution instances, on its own [25]. This information is then utilized during testing to target any specific patterns observed during the course.

Black box testing
Black box testing has no prior knowledge about the system except its interface and set of inputs for the system. The test cases are developed according to this knowledge and program’s behavior is observed on basis of possible random inputs [24]. Black box testing is usually the case of attackers who use fuzz testing to find any possible attack vector for a system and then they can build an exploit using this vector. Its advantage over white box testing is that it runs through the entire breadth of possible cases where it may find some interesting viewpoint.

Radamsa is very useful tool for black box testing. It uses a bunch of random and heuristic based fuzzing algorithms using random bit flipping, code block replacements to generate fuzz inputs. Radamsa has been successful in identifying multiple bugs in different applications [24].

Benefits
The primary advantage of fuzz testing is that it mimics the real world scenario for the software which is very difficult to do in the laboratory environment. It provides the ability to test the software with large number of nearly similar input some of which may indicate any bug or flaw which was couldn’t catch the attention of developer or security analyst. Fuzzing has been proven to find some bugs which can be serious threat to the software [24]. Smart fuzzing technique i.e. model based fuzzing is a major source of finding bugs in programs [27].
On the downside, it may be argued that fuzzing uses simple faults e.g. bit flipping, random data replacement in input, to test the program. This may not work efficiently for finding design level flaws. Fuzzing also has time complexity of exponential order i.e. $O(c^n, c>1)$ for its operation which is not practical so fuzzing programs have to make compromises for presenting results in suitable (what human care about) time which ultimately affects the performance. Since black box fuzzing uses randomness of input, this is quite unlikely that random input hit some boundary conditions. Modern fuzzing algorithm resolve this problem by using deterministic algorithms (which are out of scope for discussing here).

In case the input to a program is compressed, digitally signed, encrypted, fuzzing tool has to decompress, un-sign, decrypt the information to make fuzz test dataset and then again compress, digitally sign or encrypt the information before inputting it to the program. This procedure requires extra effort and time. Also digital signing or encryption information may not be available in some cases which can be a problem in carrying out the whole procedure.
References


