S/ OPS S

What is Fuzzing: The Poet, the Courier, and the Oracle

Summary

Fuzzing is well established as an excellent technique for locating vulnerabilities in software. The basic premise is to deliver intentionally malformed input to target software and detect failure. A complete fuzzer has three components. A poet creates the malformed inputs or test cases. A courier delivers test cases to the target software. Finally, an oracle detects if a failure has occurred in the target. Fuzzing is a crucial tool in software vulnerability management, both for organizations that build software as well as organizations that use software.

1. Fuzzing in the context of software

Fuzz testing, or fuzzing, is a type of software testing in which deliberately malformed or unexpected inputs are delivered to target software to see if failure occurs.

In this paper, we use software to mean anything that is compiled from source code into executable code that runs on some sort of processor, including operating systems, desktop applications, server applications, mobile

g a h U gb g U d dW c a hk g g m a h U f b f a Ubc UW d c f

When a piece of software fails accidentally due to unexpected or malformed input, it is a robustness problem.

In addition, a diverse cast of miscreants actively seeks to make software fail by delivering unexpected or malformed inputs. When software fails due to deliberate attack, it is a security problem.

A software failure that causes harm or death to humans is a safety problem.

Robustness, security, and safety are three faces of the same hobgoblin, software bugs. A bug is a mistake made by a developer; under the right conditions, the bug is triggered and the software does something it was not

g a h i gcd d g gic b c h cbjf g mc f hg bm a WU f U hh U i U b b g bc fl bU

1.1. Positive and negative testing

Historically, software testing has focused on functionality. Does the software work the way it's supposed to work? In functional testing, a type of positive testing, test developers create code and frameworks that deliver valid inputs to the target software and check for the correct output. For example, if we press the big red button (deliver an input), does the software turn on the city's power grid (correct output)?

In a traditional software development methodology, the software design is a list of requirements for the target software. The test development team has a fairly straightforward task of translating the design requirements into

g g h h hm gU W hk c j Uhf a g dg b f dg f Uf bh W f h b UdW W c

Functional testing is certainly important—the target software must behave as expected when presented with valid inputs. However, software that is only subjected to positive testing will fail easily when released into a chaotic and hostile world.

The real world is a mess. It is full of unexpected conditions and badly formed inputs. Software must be able to deal with other software and people who will supply poorly formed inputs, perform actions in unexpected order, and generally misuse the software. Negative testing is the process of sending incorrect or unexpected inputs to software and checking for failure.

Be aware that different negative test tools will produce different results for the same test target. Each tool works differently and will test different kinds of badly formed inputs on the target software.

1.2. Software vulnerabilities

i g g k i U b U b f c g cU b dWh kj Ug bh f i k h c g fc bdU a h fj U gf bh dW U f j c

1. Design vulnerabilities are problems with the design of the software itself. For example, a banking website that does not require users to authenticate has a serious design vulnerability. In general, design vulnerabilities must be hunted and killed by humans—automated tools simply do not exist at this level.
2. i b h gib c Ub fbc kj iU g f bh c WWh fg k gU c d h g i d W c IdU U f bm c Ud h j U f a l m U g b id kh c dU U hU U g h m hU bdgh W Uf b h U hg g b iU hb f c W fU b U h b dW U bm h j gU i h af a U g h f fhc U h g g Ugch h b c U hU bU W hi b Ub c W h g b dW Ub fbc j U f much of the seek-and-destroy work must be performed by humans.
3. i g i b g c h gj U gf U h if hh ah Dbc hj m gb g Ug U h h g dW Ub W U h W b b l c U bugs related to functionality. Negative testing, which can be heavily automated, can be used to improve the robustness and security of the software.

In addition, software vulnerabilities are unknown, zero-day, or known.

1. An unknown vulnerability is dormant. It has not been discovered by anyone.
2. A zero-day vulnerability has been unveiled by one person or a team or organization. A zero-day vulnerability is not published. The builder and users of the affected software are most likely unaware of the vulnerability. No g i l i ag g hb c dW U f U fU U fU j
3. A known vulnerability is published. Responsible vendors release new versions or patches for their software to address known vulnerabilities. While fuzzing is typically used for locating unknown code vulnerabilities, it can g i h Uig gb fh j fU m g f U i W b c cd bf h c bdW U fc

1.3. Black, white, and gray box testing

In black box testing, the test tool does not have any knowledge of the internals of the target. The tool interacts with the target solely through external interfaces.

By contrast, a white box tool makes use of the target's source code to search for vulnerabilities. White box testing encompasses static techniques, such as source code scanning as well as dynamic testing, in which the source code has been instrumented and rebuilt for better target monitoring.

Gray box tools combine black box and white box techniques. These tools interact with the target through its external interfaces, but also make use of the source code for additional insight.

n n b b U W gmU HWgl c h dU h fb f c i a gh h n n i U abi g bl hm U g h f h h bc c c f software, regardless of the availability of source code or detailed internal information. As a black box technique, fuzzing is useful to anyone who wants to understand the real life robustness and reliability of the systems they are operating or planning to deploy. It also is the reason why fuzzing is the number one technique used by black g h hU g db g d fU i h U k b Wh fc g bdU h j U f

Even without source code, the ability to more closely monitor the vitals of the target software improves the m g h h U h g h b c g g a c bi g c hdf W b g bd f Uc i fc UW a c jdf Ub Uj h bh hd f b cU U used during fuzzing to understand how the anomalous inputs are affecting the target system.

3. The oracle determines if the target has failed. Not all fuzzers implement all three parts, but to harness the power of automated fuzzing, all parts should be present.

gi gnb b a i h g g g i fgc h b c jdf cb f m U h ghi bg W f i b h iUg hc b bc g h l b

A useful fuzzer must keep records, produce actionable reports, and provide a smooth remediation process to

i g g f c df U h Wn h h f Uc b l U W

1.6. Zooming out: vulnerability management

A fuzzer will not solve all of your problems. It must be part of an arsenal of tools and part of a process for software vulnerability management. Other tools that you might also use for software vulnerability management are as follows:

- Manual security reviews
- Reverse engineering
- Static binary code analysis
- Known vulnerability scanning
- Patch management tools
- Fuzz testing

1.7. Zooming all the way out: risk management

Software vulnerability management is part of a larger picture—risk management. An organization seeking to lower, or at least understand, its overall risk will use software vulnerability management in conjunction with other risk management techniques. Fuzzing is a powerful technique for assessing the robustness and security of software, which is directly related to risk.

Now that you understand who uses fuzzing, how fuzzing relates to other software testing techniques, and where fuzzing is used in the world of vulnerability management, we will move ahead by discussing techniques and algorithms used in fuzzing.

2. The poet

n n g bg b b bn U a U m dW cbdf kU c g f h d Wn ci dU h i g g bif b chU j b b d b c

effective poet must be clever enough to craft test cases that are most likely to trigger bugs in the target software. In essence, this comes down to having a poet that creates test cases that are close to what the target expects, but malformed in some way.

The method of generating test cases has a profound effect on the quality of the test case material.

The Synopsys Universal Fuzzer (Defensics UF or DUF) is another example of an enhanced template fuzzer. DUF

h gU b i cW g Wcg Uc j g g U hg cW g U fg d h U h gc n hd bW g Uh U hU j b h c f

structure and create high-quality test cases for the target.

Corpus distillation is a method for overcoming some of the limitations of template-based fuzzing related to

m ha g h i U h e g bg U g d h hgc bf dgbU b c a Uh g h h i hkW g h h U d U g h g g h h h cU W f U

3.1. Network protocol fuzzing

One common application for fuzzing is network protocol testing. A protocol is a set of rules for how different pieces of software communicate over a network. The code that interprets protocol messages is an attack vector. Fuzzing is an excellent technique for locating unknown vulnerabilities in protocol-handling code.

Testing network software is further divided into fuzzing different roles and types of components. Many protocols include concepts of client and server, where the client initiates a connection and the server responds.

In server testing, the job of the courier is straightforward. The target software listens for incoming connections. All the courier has to do is make a connection to the target server and send a test case.

Client testing is often more complicated. The courier must act like the server, listening for incoming connections. Each time the target client makes a connection to the courier, the courier responds with a test case. Usually providing unexpected or malformed inputs to the UI. For example, in an application on a traditional desktop, a

5. Wrap up

Fuzzing is an excellent technique for locating vulnerabilities in software. The basic premise is to deliver intentionally malformed input to target software and detect failure. A complete fuzzer has three components. A poet creates the malformed inputs or test cases. A courier delivers test cases to the target software. Finally, an oracle detects target failures.

i n n i hb g b h f g b W eiU b U j hb n n hU bW g b Wc h b ga hWj h gc h fi cU h d f c d
more effective when it is able to create test cases that are almost correct, but anomalous in some way. Different oracle techniques provide varying levels of failure detection capability. Multiple oracle techniques can be used together to help detect the maximum number of failures.

bugs earlier rather than later.

Fuzzing is a crucial tool in software vulnerability management, both for organizations that create software as well as organizations that use software. Fuzzing must be deployed as part of a process. Builders use fuzzing as
b hb U h U GU f a Ud cf W hbfm j k c g di g m n nW gi b f U bl bUW f W lc c b j bU f WU c
h g bU jUm b c U n ng U bgW U a b gl a m i U b gn j i g c a g g h Ud g W i l gb W h l d j c U
i h hg U b h f U fU h h fl U U a m f c hb f g c d h c fi c df Wbg f U f c hU h U h U U fc
and possibly exploited in production scenarios can be hugely expensive.

bh g hbUa hc f a dW b c Uf hbU i b b g b l bU i km h n n U h c dfg Wj b h dnjdf b c df
against other types of damage. If a bug in your product leads to a catastrophic failure or a massive data breach, your reputation might not recover and your legal liability could be insurmountable. If you provide a service, such as healthcare, power, communications, or another critical infrastructure, bugs in the products you're using could lead to human harm or environmental damage.

i b b g gb l g bU a mi g Ul b j gm h c g i c dfh i W m bfW c U fh h a c b g bUg fmg b d Ucg b U U W U
lives.

Explore how the Synopsys Fuzz Testing tool can help you build more secure software.

Learn more.