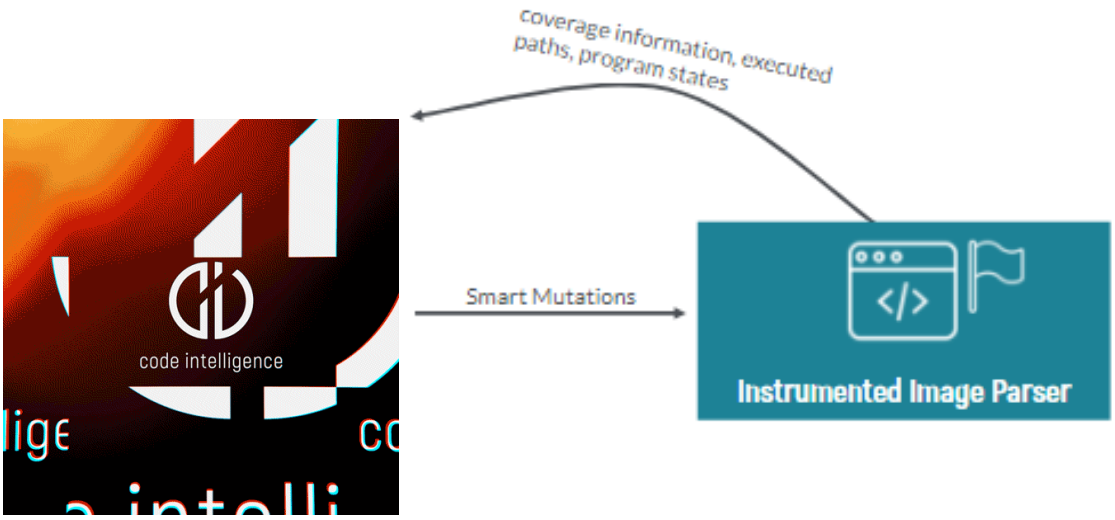
code intelligence

Fuzzing Fact Sheet

How Fuzz Testing Enables Developers to Ship Secure Software Fast

What is Fuzz Testing?

Fuzz Testing is a dynamic testing method used for finding functional bugs and security issues in software. During a fuzz test a program is fed with invalid, unexpected, or random inputs, with the aim to crash the application. This is currently one of the most effective approaches to automatically detect bugs and vulnerabilities in software.



Fuzzing Explained on the Example of an instrumented Image Parser

With Fuzzing You Can Protect Your Code Against the Unexpected

Modern fuzzers executes a program with invalid, unexpected, or random inputs. This way your security tests can cover unlikely or unexpected edge cases, that would not be detected with other testing approaches.

With Fuzzing Each Finding Leads to More Findings

Once a fuzzing solution found an input that has caused a crash, it uses mutation algorithms to generate even more inputs which can reproduce the finding with a high probability.

With Fuzzing Developers Can Automate Their Security Testing at Scale

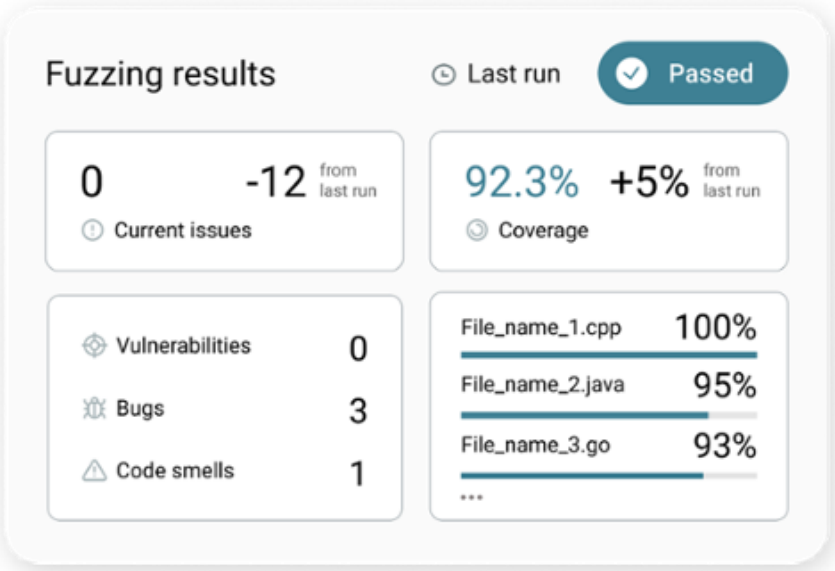
Modern fuzzing solutions can automatically analyse the structure of the code they test. They use this information to generate thousands of test cases per second, and they also mark each path the inputs take through the program. This way a fuzzer gets details feedback about the code coverage, the inputs are reaching during the execution of the code.

With Fuzzing You Can Run Automated Security Tests With Each Pull Request

If you run fuzzing in your CI/CD pipeline, you can perform automated penetration tests with each pull request. This helps you fix bugs faster and to improve the quality of your code. See fuzzing in action.

With Fuzzing You Can Achieve Maximum Code Coverage Without False Positives

Because fuzzers execute the software under test, they always provide you inputs that you can use to reproduce bug. That's why fuzzing enables you to reach up to 99% code coverage without any false positives. Click here to see an example of a full bug report from a fuzzing project.



CI Fuzz Coverage Reporting

Fuzzing Explained for Developers

Due to their high degree of automation, modern fuzzing solutions, such as CI Fuzz, enable developers to conduct advanced security tests themselves. For more detailed information read this instruction: Fuzzing 101

Industries Where Fuzzing Is Used

More and More Industry Standards Require Fuzz Testing

Due to increasing security regulations, more and more software companies must run automated security tests before shipping their software. That is why many industry and ISO standards recommend integrating automated fuzz testing into the development process. Especially in industries, that already have advanced quality and security regulations. Good examples are ISO/SAE 21434 and UNECE WP.29, which deal with the security of automotive software.

What Standards and ISO Norms Recommend Fuzzing?

ISO 26262
Road vehicles – Functional Safety

UNECE WP.29
United Nations World Forum for Harmonization of Vehicle Regulations

ISA/IEC 62443-4-1
Secure Product Development Lifecycle Requirements

ISO/SAE DIS 21434
Road Vehicles — Cybersecurity Engineering

UL2900-1 and **UL2900-2-1**
Healthcare and Wellness Systems - Software Cybersecurity for Network-Connectable Products

ISO/IEC/IEEE 29119
Software and Systems Engineering - Software Testing

ISO/IEC/IEEE 29119
Software and Systems Engineering - Software Testing

ISO/IEC 12207
Systems and Software Engineering – Software Life Cycle Processes

ISO 27001
Information Technology – Security Techniques – Information Security Management Systems

ISO 22301
Security and Resilience — Business Continuity Management Systems

IT-Grundschutz (Germany)
Based on ISO 27001

and others

How Fuzzing Prevented a Total Ethereum Shutdown

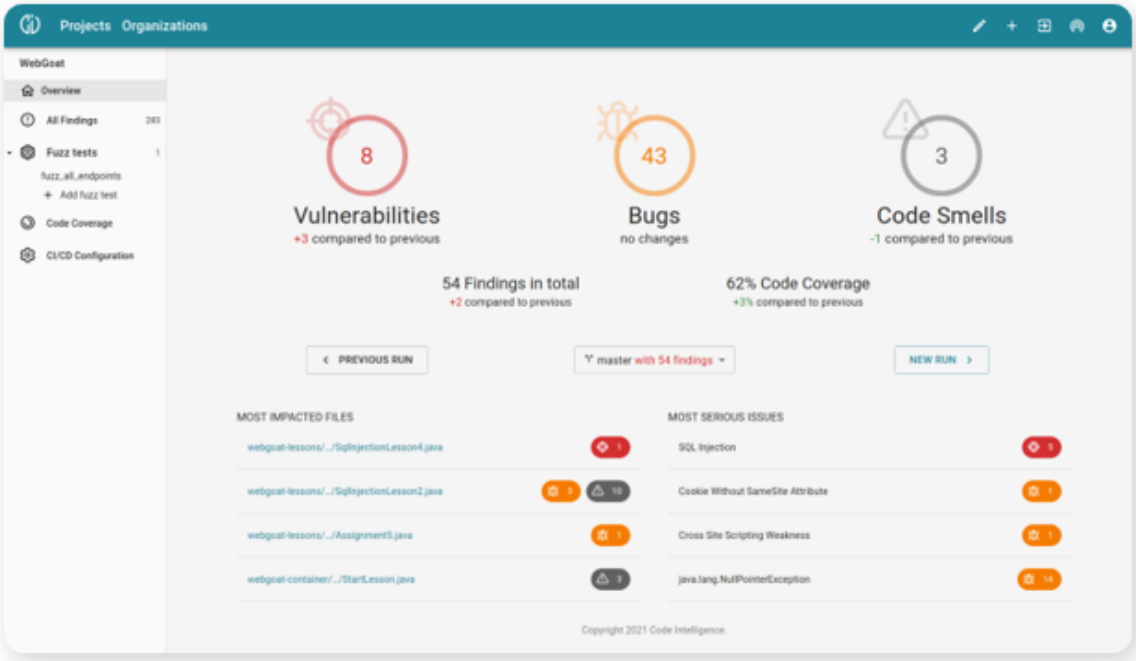
Fuzz testing finds bugs that other testing methods cannot detect. In November 2020 a serious DoS vulnerability was fixed in the source code of the Ethereum network due to advanced fuzz tests. In the wrong hands, this vulnerability (CVE-2020-28362) could have caused the shutdown of the entire Ethereum network. Although the memory safe Golang module has already undergone extensive security testing, this vulnerability could only be found through fuzz testing.

Frequently Asked Questions About Fuzzing (FAQs)

1. Why is Fuzzing (Especially) Useful for Security Testing?

There are some features that make fuzzing enormously useful for security testing.

- With Fuzzing you can reach maximum code coverage without false positives.
- Fuzzing provides developers with reliable and reproducible testing inputs that caused the application to crash. This helps them to find and fix bugs faster.
- Fuzzing is an almost completely automated testing technique.



CI Fuzz Reporting Dashboard

2. What Is Feedback-Based Fuzzing?

Modern fuzzing engines use smart algorithms to increase the amount of code. The commonly used term for this is feedback-driven or feedback-based fuzzing. Measuring code coverage, the fuzzer can monitor which parts of program were reached with a given input.

3. What Is a Fuzz Target?

Fuzz targets are small programs that test predefined API functions, similar to unit tests. However, the inputs are not provided by the developer but produced with **fuzz generators**.

The fuzz generators are responsible for creating random mutations of inputs that are sent to the software under test (SUT). The **delivery mechanism** processes inputs from fuzz generator and feeds them to the SUT for execution.

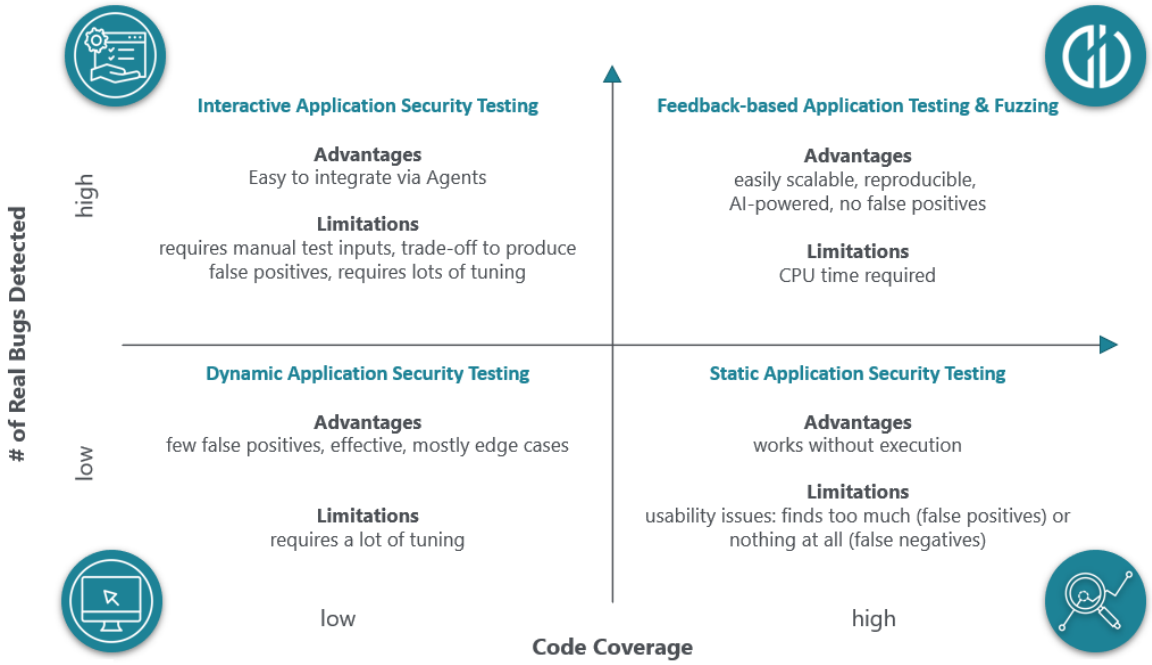
Finally, the **monitoring system** keeps track of how inputs are executed within the SUT. It detects triggered bugs, which plays a critical part in the fuzzing process, as it also influences what types of vulnerabilities can be discovered during fuzzing. Click here to learn how to set up fuzz targets yourself.

```java
public class ExampleValueProfileFuzzer {
  // consume data from fuzzer and give it to function under test
  public static void fuzzerTestOneInput(FuzzedDataProvider data) {
    String str = data.consumeString(100); // get a string with max_len 100
    long input1 = data.consumeLong();
    long input2 = data.consumeLong();
    // call function under test with fuzz data
    ApplicationUnderTest.consume(str, input1, input2);
  }
}
```

Example of a fuzz target, for Java applications. See full gist

4. What Are the Benefits of Fuzzing Compared to Other Testing Methods?

If you are looking for a way to secure your software, there are a variety of testing approaches, such as Static Applications Security Testing (SAST), Dynamic Application Security Testing (DAST), Interactive Application Security Testing (IAST), and Feedback-based Application Security Testing (FAST). Each of these methods has its advantages and disadvantages. We have compared some of them in the table below.



Fuzz Testing compared to other Application Security Testing approaches, such as SAST, DAST and IAST

5. What Bugs Can You Find with Fuzzing?

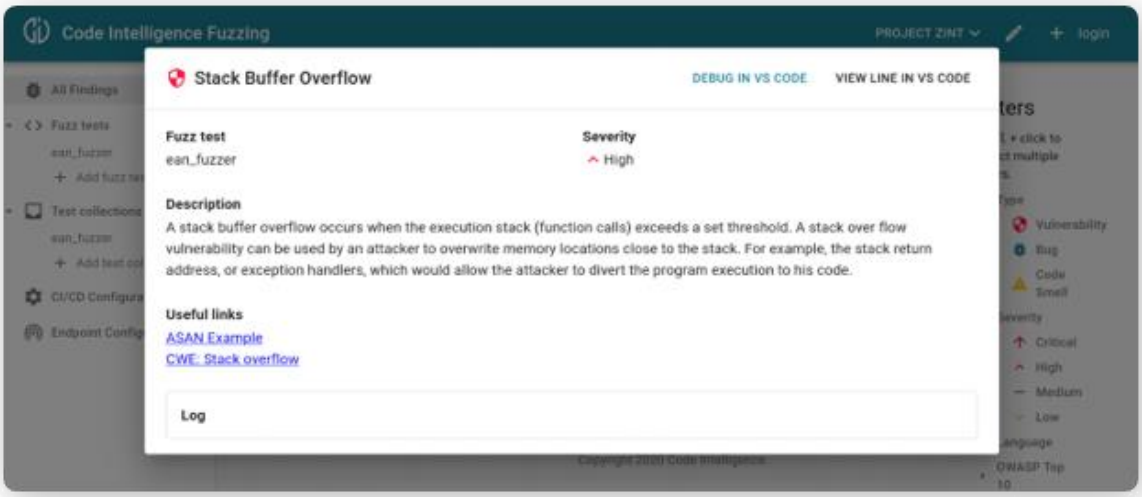
Since 2018, Code intelligence provides a platform for automated fuzz testing. Working closely together with industry and academia, the engineers at Code Intelligence were able to find thousands of bugs and extended the reach of modern fuzz testing to a variety of different use cases. Here, you will find an overview of some of the bug classes that the CI team found over the past years.

Memory Leaks – incorrect management of memory allocation leads to an exhausting available memory.

Injections – untrusted input supplied to a program.

Sensitive Data Exposure - an application inadvertently exposes personal data.

Buffer Overflows - exceeding the buffer when writing data ends up with overwriting adjacent memory locations.



Example of an error message in CI Fuzz

Use After Free – not pointing to a valid object result in data corruption, segmentation faults or general protection faults.

Data Races - accessing the same memory location concurrently in multi-thread process that is related to security vulnerabilities.

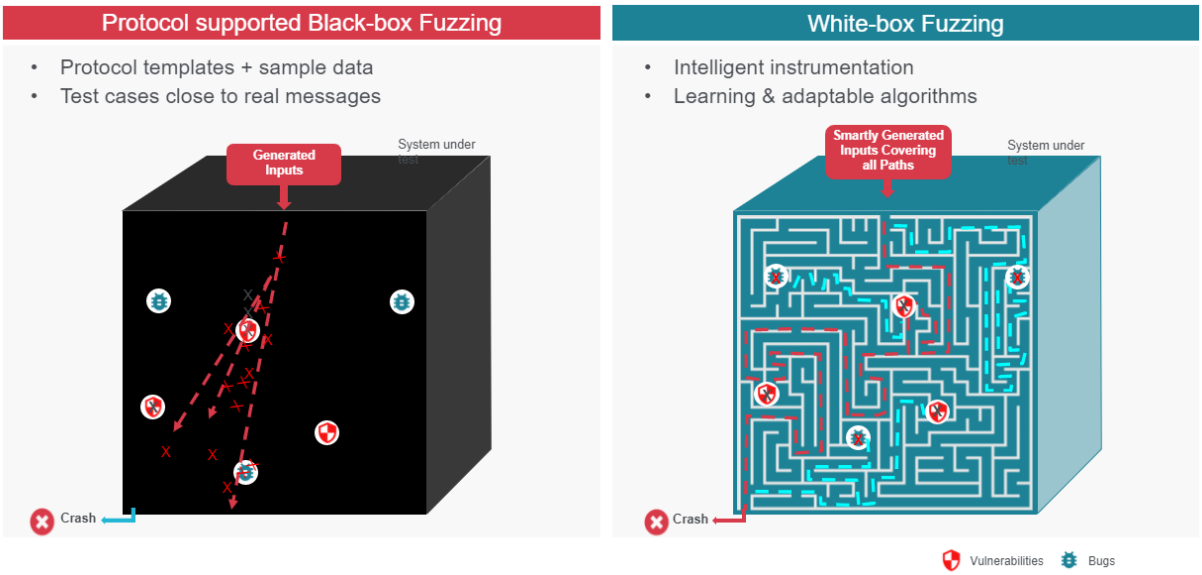
Software Crashes - failing to function properly and exiting, possibly fatal system errors.

However, there are many more bug classes, that you can find with fuzzing.
Click here to see the full list.

6. Black-Box Fuzzing vs. White-Box Fuzzing

Black-box fuzzing generates inputs for a target program without knowledge of its internal behavior or implementation. A black-box fuzzer may generate inputs from scratch, or rely on a static corpus of valid input files to base mutations on. Unlike coverage-guided approaches and white-box fuzzing, the corpus does not grow here.

White-box fuzzing analyses the internal structure of the program. With each new fuzzing run, they learn to track and maximize the code coverage. White-box fuzzers usually use intelligent instrumentation and adaptable algorithms, which makes them more effective and accurate in detecting vulnerabilities.



The difference between black-box-fuzzing and white-box-fuzzing

7. How to Get Started With Fuzzing?

Try to start with a simple open-source fuzzer like Atheris (for Python) or Jazzer (for Java). These open-source tools can help you to get comfortable with this testing approach. But if you want to try fuzzing in a more complex environment, there are a number of easy to use enterprise solutions, such as CI Fuzz, which comes with many additional integrations and features, like OWASP vulnerability detection, automated bug reporting, CI/CD and dev tool integration, or API fuzzing.

8. List of Common Open Source Fuzzing Tools

Developers can benefit from a whole range of open-source fuzzing tools. There are often specialized for specific use cases (e.g. Kernel fuzzing) or programming languages.

AFL++ - American fuzzy lop file format fuzzer
libFuzzer - in-process, coverage-guided fuzzing engine for targets written in C/C++
Jazzer - coverage-guided fuzzer for Java, Kotlin, and Clojure
Honggfuzz - general-purpose, easy-to-use fuzzer with interesting analysis options.
OSS-Fuzz - continuous fuzzing for open-source software
ClusterFuzz - scalable fuzzing infrastructure that finds security and stability issues.
Radamsa - test case generator for robustness testing
BFuzz - an input based fuzzer tool
KernelFuzzer - cross platform kernel fuzzer framework
Go-fuzz - a library for populating go objects with random values

Click here to see more task-specific open-source fuzzers.

9. Autofuzz: Fuzz Your First Application Today

Most fuzzers require users to create fuzz targets or test harnesses manually. However, to the ease fuzzing integration, some open-source fuzzers like Jazzer (Java Fuzzer) provide an autofuzz mode, making the creation of fuzz targets obsolete.

```
jazzer --cp=commons-compress-1.20.jar \
       --autofuzz=org.apache.commons.compress.archivers.sevenz.SevenZFile::new
```

Example of a simple autofuzz command with Jazzer

This way fuzzing becomes as simple as writing unit tests. With Jazzer, every developer can now to secure a Java library in less than 3 minutes, which makes this kind of open-source fuzzing perfect to gain first experiences with fuzz testing.

Ty out Jazzer yourself, read this blog post or watch this Jazzer tutorial.