Zero-Day Vulnerabilities and Fuzzing: From Threat Analysis to Exploit Discovery

Maria Nefeli Ntantouri

I. Introduction

MOST software written includes design and implementation weaknesses. These weaknesses can create vulnerabilities that attackers may exploit. Zero-day vulnerabilities constitute one of the most critical threats in the cybersecurity landscape. A zero-day attack is a cyber attack that exploits a vulnerability that has not been revealed publicly. Attackers exploit these vulnerabilities before developers have the opportunity to defend the system [1], often resulting in severe breaches. Antivirus programs cannot detect the attack. Data on zero-day attacks can be discovered only after the attack has been executed, therefore, it is hard to study them. The vulnerability life cycle, from discovery to patching, is also complicated and highly variable depending on factors like software type, attacker motivations, and disclosure practices, according to extensive vulnerability analysis [2]. To improve overall cybersecurity resilience and create effective defensive strategies, it is essential to comprehend the nature and exploitation of zero-day vulnerabilities. The goal of this study is to investigate how zero-day vulnerabilities are found, how they are exploited, and what detection and mitigation strategies are available.

II. Theoretical Background

A. Definition and Lifecycle of Zero-Day Vulnerabilities

A zero-day vulnerability is a security flaw in a system, device, or software that is unknown to the party responsible for its remediation [1]. Since there is no fix or public knowledge, these vulnerabilities offer attackers a unique opportunity to exploit targets without resistance. The life cycle of a zero-day vulnerability generally consists of seven phases. In the discovery phase, a vulnerability is found by ethical researchers, attackers, or vendors. The discovery date usually becomes publicly known after the disclosure of the vulnerability. The weaponization phase, where malicious actors develop exploit tools or malware that leverage the vulnerability. The exploitation phase includes active use of the vulnerability to gain unauthorized access, cause disruption, or steal information. Eventually, security systems or researchers identify unusual behavior or discover the exploit during investigations, and this constitutes the detection phase. Following detection comes disclosure. The vendor is notified of the vulnerability, or it becomes public. At this stage, the vendor develops a patch and releases it (patch development phase). The remediation phase completes the life cycle after organizations update systems and apply patches to prevent vulnerability exploitation. RAND Corporation implemented a study based on rare access to a dataset of more than 200 zero-day vulnerabilities. According to their findings, zero-day vulnerabilities have an average life expectancy of 6.9 years [3]. This time frame includes the initial private discovery until the public disclosure. The kind of vulnerability, the popularity of the impacted software, and the speed at which patches are released following disclosure all have a significant impact on zero days [2]. Zero-day vulnerabilities can be used as a weapon for geopolitical ends or to inflict extensive financial harm, as demonstrated by real-world attacks such as Stuxnet. One of the earliest known cyberweapons to attack Iran's industrial control systems simultaneously using several zero-day vulnerabilities was Stuxnet (2010). The malicious computer worm, which has been around since at least 2005, is thought to have seriously harmed Iran's nuclear program by taking advantage of flaws in supervisory control and data acquisition (SCADA) systems [4]. Another real-life example is the EternalBlue vulnerability from 2017, which was based on a zero-day exploit leaked from U.S. National Security Agency (NSA) tools. It gained widespread notoriety when it was used in the WannaCry ransomware attack and later in the NotPetya malware. Microsoft had managed to issue a patch a few days before the massive attack, but thousands of machines were left unprotected. Despite the patch being available, the large number of vulnerable systems showed the importance of rapid patching after a zero-day release[5].

B. Zero-Day Discovery Techniques

The discovery of zero-day vulnerabilities is a critical and complex process in cybersecurity, which is based on a combination of static and dynamic analysis techniques. Static analysis, through tools like Coverity [6], allows the inspection of source or binary code without execution, helping to identify suspicious patterns or insecure constructs[7]. In contrast, dynamic analysis, performed with sandboxing tools like Cuckoo, aims to study software behavior under real execution conditions. To be effective, the target program must be executed with sufficient test inputs to address the ranges of possible inputs and outputs [8]. Among the most efficient and automated methods is fuzzing, where tools such as AFL, a free software fuzzer [9], generate mass inputs, causing errors or crashes that reveal unknown vulnerabilities [10]. Complementarily, techniques based on heuristic rules and patterns provide basic detection, but fall short in cases of unprecedented attacks. Finally, the use of machine learning has begun to be applied in anomaly detection environments, although its practical value depends on the quality of the training data. No single technique is completely sufficient on its own, so a layered combination is necessary to effectively address zero-day threats.

1) Static Analysis: It refers to the examination of source code or binary executables without actually running the program. This technique allows security researchers to uncover potential vulnerabilities by examining the control flow, data flow, and structural properties of the software. It is particularly valuable in the early stages of vulnerability research, as there is no risk of damaging the system being analyzed. Popular tools such as IDA Pro and Ghidra are widely used to disassemble and decompile binaries, providing a detailed view of low-level operations even in the absence of source code. One of the main advantages of static analysis is its ability to systematically cover all code paths, making it ideal for identifying insecure patterns such as buffer overflows or use-after-free vulnerabilities. However, static analysis is not without its limitations. When dealing with obfuscated, packed or proprietary code, the accuracy of the analysis can be significantly reduced. In such cases, understanding the true program logic becomes difficult, and the output may be cluttered with false positives or misleading information [11]. Despite these limitations, static analysis remains a cornerstone technique in vulnerability discovery, especially when combined with dynamic methods.

2) Dynamic Analysis: This technique entails monitoring the behavior of a software program while it is executed in a controlled environment. Unlike static analysis, this approach provides insight into how an application interacts with system resources, memory, and external inputs at runtime, making it particularly effective at detecting vulnerabilities that only manifest themselves during execution [12]. Tools such as Cuckoo Sandbox allow analysts to run potentially malicious code in isolated virtual environments and monitor system calls, file modifications, network activity, and registry changes without compromising the integrity of the host machine [13]. Similarly, Valgrind enables in-depth memory usage analysis, helping to identify issues such as memory leaks, invalid read/write operations, and undefined behavior [14]. Using sandbox environments is a fundamental component of dynamic analysis, providing a safe space to execute suspicious or unknown binaries while collecting telemetry data. One of the major strengths of dynamic analysis is its ability to identify runtime behaviors, such as just-in-time code generation, conditional branching, or polymorphic code execution, that would be difficult or impossible to detect statically. However, its effectiveness can be limited by code paths that are not executed during analysis, making coverage of test inputs a critical factor in the accuracy of results.

3) Heuristics and pattern matching: A traditional but widely used approach to detecting software vulnerabilities and malicious behavior. This method depends on analyzing known patterns or signatures that are typically linked to suspicious or malicious code constructs [15]. By comparing code or program behavior against signature databases, such as those maintained by antivirus engines, it is possible to flag known threats with high speed and accuracy. These techniques are often implemented in commercial anti-virus software, intrusion detection systems (IDS), and static analyzers, providing effective protection against previously identified malware and vulnerabilities. In addition, heuristic rules can be created to detect code that exhibits behavior statistically associated with exploits, even if it doesn't exactly match a known signature. However, the main limitation of this approach is its reduced ability to detect unusual or zero-day vulnerabilities, as these do not yet fit any existing pattern. While heuristics can sometimes detect anomalies that resemble known threats, their effectiveness is greatly reduced in the face of innovative or obfuscated attacks. As such, pattern matching is best used as part of a layered defense strategy, complementing more adaptive or behavior-based methods such as dynamic analysis or fuzzing.

4) Machine learning: ML has entered the modern cybersecurity world, especially in environments where the traditional signature-based techniques are inadequate. The main goal is to use ML models, such as classifiers or anomaly detection algorithms, to analyze and find patterns of system behavior. A detection of a deviation can imply malicious activity [16]. Usually, the supervised models get trained on labeled datasets based on known malicious behavior. Unsupervised methods, such as clustering or autoencoders, can be used to detect unknown anomalies without prior labeling. Examples include the use of classification algorithms to distinguish between normal and exploit traffic or anomaly detection models to highlight unexpected system calls or memory usage patterns.

The main advantage of ML is the ability to generalize and identify threats that have not been detected before but have characteristics similar to already known malicious behaviors. However, the outcome highly depends on the quality and quantity of the data used in the training of the model. Collecting representative data and correctly labeled sets is challenging. Unbalanced classes can also negatively affect the training. Apart from those challenges, ML systems are also vulnerable to adversarial attacks, where malicious actors manipulate input to trick the model. Despite these challenges, ML techniques offer a powerful tool, especially when used in combination with other methods such as dynamic analysis and fuzzing, as part of a layered defense strategy against zero-day attacks.

Table I presents a comparative overview of the main zero-day vulnerability detection techniques, based on key criteria such as speed, accuracy, resource requirements, and the ability to detect unknown vulnerabilities.

TABLE I
COMPARISON OF ZERO-DAY VULNERABILITY DETECTION TECHNIQUES

Method	Speed	Accuracy	Resources	New Threats
Static	High	Medium	Low	Low
Dynamic	Medium	High	High	Medium
Fuzzing	High	High	Medium	Very High
Heuristics	High	Medium-High	Low	Very Low
ML	Medium	Medium	High	High

C. Exploitation techniques

This subsection explores how attackers identify, market, and exploit zero-day vulnerabilities, focusing on the phases

of the attack. Zero-day vulnerabilities are extremely valuable targets for attackers, as they offer them the ability to carry out attacks without being immediately detected by traditional defense systems. Their exploitation involves an organized chain of actions, starting from the acquisition or discovery of the vulnerability, weaponization - that is, its transformation into a functional exploit - and its delivery to the target through methods such as phishing or drive-by downloads[17]. After successful exploitation, attackers focus on maintaining their presence in the system and avoiding detection using sophisticated evasion techniques. The impact of such attacks can be devastating, both economically and sociopolitically, as has been mentioned in cases such as Stuxnet and WannaCry[18].

1) Acquiring the Zero-Day Vulnerability: The first and crucial step in the process of exploitation by an attacker is to obtain the vulnerability. Such vulnerabilities are often identified privately by independent hackers (also known as security researchers), government agencies, or even cybercriminal groups that specialize in finding software vulnerabilities. Once a vulnerability is identified, it can be exploited directly by the discoverer or sold to third parties through zero-day exploit markets on the dark web or specialized zero-day marketplaces such as Zerodium or Exodus Intelligence [19]. The price of such vulnerabilities depends on the platform, the severity, and the ability to execute code remotely. For example, the value of a full remote jailbreak on iOS, which uses a privilege escalation exploit to remove software restrictions imposed by Apple on devices running iOS, can exceed $1 million [20]. This ecosystem also includes so-called ”vulnerability brokers”, who act as intermediaries between researchers and government or private entities, managing both the negotiation and legal coverage of the purchase.

2) Weaponization: Once a zero-day vulnerability has been acquired, the next stage is weaponization, or turning it into a functional exploit that can be embedded into malicious tools or software. At this stage, attackers develop specific payloads that exploit the vulnerability to achieve goals such as remote code execution, system compromise, or data extraction. Often, exploits are included in widely used exploit kits such as RIG and Neutrino, which make it easier for even unsophisticated attackers to exploit vulnerabilities through automated attacks [21]. A notable example of weaponization is the previously mentioned EternalBlue exploit.

3) Delivery & Exploitation: After deploying a zero-day exploit, the delivery and exploitation phase occurs, where attackers deliver the exploit to the target and activate its malicious functionality. The most common delivery methods include phishing emails with malicious attachments or links, malvertising (advertisements containing malicious code), and drive-by downloads, where simply visiting a compromised website is enough to trigger the attack. Once the exploit is successfully executed on the victim's system, the payload is activated, which can install backdoors, ransomware, or provide remote access to the attacker [22]. The consequences of a successful exploit include remote code execution, privilege escalation, or even a complete system breach. Many factors, such as the delivery method, the security level of the target, and the exploit's ability to bypass protections such as ASLR (Address Space Layout Randomization) or DEP (Data Execution Prevention), can have an impact on the effectiveness of this phase.

4) Persistence & Avoidance: After the initial exploitation of a zero-day vulnerability, attackers aim to maintain their presence in the system and avoid detection by security mechanisms. For persistence, techniques such as creating scheduled tasks, modifying registry entries, or installing rootkits are used to allow the malicious code to execute at each system reboot. At the same time, evasion techniques are used, such as using encryption to hide the payload, polymorphism so that the code changes form each time it is executed, and anti-sandbox techniques where the malicious code delays activation or tests to see if it is running in a virtual environment to avoid analysis [23]. The combination of these methods makes zero-day attacks extremely difficult to detect and allows attackers to remain undisturbed for long periods of time, gathering information, executing additional attacks, or preparing for future escalation.

5) The Consequences: Exploiting zero-day vulnerabilities can be devastating, affecting not only technological security but also economic and geopolitical stability. At an operational level, a successful attack can cause downtime, data loss, intellectual property leakage, and severely undermine customer and partner confidence. Financially, the cost per incident can be in the millions of dollars, including damages, compliance fines, and remediation costs. Geopolitically, zero-day vulnerabilities are also being used as a means of cyber espionage or cyber warfare, with targets including government agencies, critical infrastructure, and strategically sensitive organizations. A prominent example is Sandworm Group activities, a Russian-linked Sandworm hacker group which has a history of exploiting zero-day vulnerabilities for cyber warfare. Notably, in 2014, they used a zero-day in Microsoft Windows to target Ukrainian government entities. In 2015, they launched a cyberattack on Ukraine's power grid, disrupting electricity supply to approximately 230,000 residents [24]. Such incidents highlight the seriousness and multi-dimensional nature of the consequences of exploiting unknown vulnerabilities.

D. Defense and Mitigation Techniques

The nature of zero-day attacks, which exploit unknown and unpatched vulnerabilities, makes defending against them particularly challenging. As traditional detection techniques, such as signature-based detection [25], are ineffective, organizations are turning to more sophisticated prevention and mitigation strategies. Methodologies such as behavioral analysis, sandboxing, threat intelligence sharing, and deception technologies are key tools in the cybersecurity arsenal. These techniques, combined with memory protection measures and temporary solutions, such as virtual patching, allow defenders to detect and mitigate zero-day attacks even before they become publicly known.

1) Signature-based vs. Behavior-based Detection: Traditional attack detection techniques, such as signature-based intrusion detection/prevention systems (IDS/IPS), rely on known patterns of malicious code or activity (signatures) to detect

threats [26]. However, this approach has serious limitations in the case of zero-day vulnerabilities, as these signatures have not yet been created or captured. As a result, such attacks can completely evade detection. In contrast, behavior-based or anomaly-based techniques offer a more dynamic and adaptive defense. They are based on the analysis of system behavior or network traffic to detect deviations from the 'normal' operating profile. They use artificial intelligence, machine learning, and heuristics to identify suspicious actions, even when there is no prior knowledge of the specific exploit [16]. Examples include tools such as Suricata and Zeek, which support the integration of machine learning-based plug-ins for improved threat detection and classification. While more flexible, behavior-based techniques require proper training and configuration to avoid false positives.

2) Sandboxing & Emulation: Sandboxing is an important proactive measure for analyzing and controlling unknown or potentially malicious software without compromising the actual system. It isolates the suspicious executable and runs it in a virtual, controlled environment where its behavior can be observed and recorded [27]. Tools such as Cuckoo Sandbox allow automatic monitoring of operations such as file changes, registry entries, network communications, or privilege escalation attempts. At the same time, emulation can be used to simulate different environments and operating systems, extending the range of possible execution scenarios. The use of sandbox environments provides the ability to detect malicious behavior early, preventing the spread of a zero-day exploit to the production system. However, some sophisticated attackers use sandbox evasion techniques, such as delayed execution or checking for the existence of a virtual environment, which requires continuous improvement.

3) Threat Intelligence & Information Sharing: The collaborative process of organizations exchanging cyber threat intelligence to collectively enhance their defenses against cyberattacks is a strategic method for early detection and defense against zero-day attacks. By collecting and analyzing data from multiple sources, such as the Malware Information Sharing Platform (MISP) [28] or the Open Threat Exchange (OTX) [29], organizations gain access to information such as new attack techniques, attacker tactics, and Indicators of Compromise (IoCs). The continuous flow of this information enables rapid identification of anomalous behavior and immediate response to potential attacks before they become widely known. In addition, collaboration between governments, private companies, and security communities creates a collective defense that reduces the window of vulnerability and increases resilience to advanced threats. However, the effectiveness of threat sharing depends on the quality, speed, and reliability of the data in transit, as well as the trust between participating actors.

4) Virtual Patching & Memory Protection: When a formal security fix for a vulnerability is delayed or unavailable, organizations turn to temporary protection techniques such as virtual patching. This approach is typically implemented through web application firewalls (WAFs) or intrusion prevention systems (IPSs), which analyze incoming and outgoing traffic in real time and block malicious actions that attempt to exploit known or unknown vulnerabilities. Virtual patching provides a fast and non-intrusive defense, particularly useful on systems that cannot be disrupted for traditional patching [30].

At the same time, basic memory protections such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) make it much harder to exploit vulnerabilities at the operating system level. ASLR changes the layout of memory each time a program is executed, making attacks based on known memory addresses unreliable. DEP, on the other hand, prevents code from executing in areas of memory reserved for data, thus preventing buffer overflow attacks. While these techniques do not eliminate risk, they significantly reduce the attack surface and make zero-day exploits more difficult.

5) Deception Techniques: The use of deception techniques is a powerful mechanism for preventing and detecting zero-day attacks by deceiving attackers and revealing their behavior. The main technique in this category is the use of honeypots: virtual systems or networks that simulate vulnerable infrastructure to lure attackers into attacking them without compromising real systems [31]. These traps record in detail the actions of attackers, such as the tools and techniques (TTPs - Tactics, Techniques, Procedures) they use, providing invaluable information for identifying new attacks and improving defenses.

In addition to honeypots, honeytokens - virtual data or files designed to be valuable to the attacker to detect interaction with them - are also used. These technologies not only help to monitor and capture attacker activity, but also help to prevent further attacks by identifying the TTPs used, allowing the system's defenses to be improved.

III. Fuzzing: Theory and Practice

Fuzzing is an automated software testing method that identifies vulnerabilities by supplying a program with a large volume of unexpected or random input data. The basic idea behind fuzzing is that when a program is exposed to abnormal or unpredictable data, errors can occur that reveal underlying vulnerabilities, some of which may be zero-day.

1) What is Fuzzing: Fuzzing is a software testing technique that aims to identify security bugs or glitches by providing the System Under Test (SUT) with invalid, unexpected, or random input data. The aim is to create situations that the developer did not anticipate in order to reveal vulnerabilities such as buffer overflows, memory leaks, or unprotected exceptions. Fuzzing can be applied to either compiled binaries or source code, depending on the technique used. There are three main categories of fuzzing, depending on the level of information the tool has about the target software: Black-box, White-box, and Gray-box fuzzing. In black-box fuzzing, the tool ignores the internal structure of the software and simply sends inputs in bulk and monitors the outputs for signs of failure. It is easy to implement, but can be inefficient on complex systems. White-box fuzzing has full access to the source code and can use techniques such as static analysis or symbolic execution to produce more targeted and efficient input. Although powerful,

it requires more resources and technical knowledge. Grey-box fuzzing is somewhere between the two previous techniques. The tool has partial knowledge of software behavior (usually through feedback such as code coverage) and guides inputs to improve coverage. The popular AFL (American Fuzzy Lop) tool is a typical example of grey-box fuzzing and is considered an industry standard due to its efficiency. Choosing the right technique depends on the software used, the resources available, and the desired depth of analysis.

2) Fuzzing Tools: In the field of fuzzing, several tools have been developed to automate and optimise the process of generating random or targeted inputs. Three of the most popular fuzzing tools are AFL (American Fuzzy Lop), LibFuzzer and BooFuzz. AFL is a grey-box fuzzing tool that uses coverage-guided fuzzing techniques. It analyzes the coverage of code at runtime to guide the creation of new test cases. It is extremely effective and has discovered many vulnerabilities in real systems. LibFuzzer is based on the LLVM compiler infrastructure. It is an in-process, coverage-guided fuzzing tool that integrates directly into unit tests. Although it requires modification of the source code for integration, it offers high efficiency and fast execution. Finally, BooFuzz is a black-box fuzzing tool based on the Sulley framework. It is suitable for fuzzing protocols and network targets. Although it doesn't use code coverage information, it is extremely useful in cases where we don't have access to the target code. Table II summarizes the main features and differences between the AFL, LibFuzzer and BooFuzz fuzzing tools.

TABLE II
FUZZING TOOLS

Feature	AFL	LibFuzzer	BooFuzz
Fuzzing Type	Grey-box	White-box	Black-box
Code View	Standard execution	Requires access to the code	Does not require access
Coverage-Guided	✓	✓	✗
Installation	Medium	Complicated for beginners	Easy
Specialty	Binary files, CLI apps	Unit tests, libraries	Network protocols

IV. FUZZING DEMO WITH BOOFUZZ

A. Tool Description and Setup

BooFuzz is an open-source black-box fuzzing tool designed to help security researchers find vulnerabilities by automatically sending crafted inputs to a target, monitoring for crashes or unwanted behavior. It is the successor to Sulley and is widely used for testing socket services, protocols, CLI applications, or even custom services. Its installation is simple and is done via pip:

```
pip install boofuzz
```

The only requirement is to have Python 3.x installed. Once installed, it can be used either through a custom Python script or by modifying examples provided in the official documentation of the tool. The fuzzing script consists of a description of the target, the connection method (e.g., TCP socket), and the input fields to be fuzzed.

B. Fuzzing Scenario

As part of this section, a practical fuzzing scenario was set up to identify crashes in a vulnerable socket server. The scenario was executed using the Boofuzz tool, an open-source library for automated fuzz testing in application-layer protocols. This was performed on an Ubuntu virtual machine using Python 3 and in a virtual environment.

1) Implementation Environment: Boofuzz was installed in a remote Python virtual environment as follows:

```
sudo apt install python3-venv -y
python3 -m venv fuzzenv
source fuzzenv/bin/activate
pip install --upgrade pip
pip install boofuzz
```

2) Vulnerable Socket Server: The target of fuzzing is a simple TCP socket server listening on port 9999. The server stops execution when it receives the special character sequence $b'CCCCCC'$, which simulates a crash condition. The relevant server snippet is shown below:

```
HOST = '127.0.0.1'
PORT = 9999
SECRET_SEQUENCE = b'CCCCCC'

def start_server():
    with socket.socket(socket.AF_INET,
    socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen(5)
        ...
        while True:
            conn, addr = s.accept()
            with conn:
                while True:
                    data=conn.recv(1024)
                    if SECRET_SEQUENCE
    in data:
                        print("[!]
    Secret sequence received. Shutting
    down.")
                        return
```

The server terminates normally with the return, which means that the whole application is interrupted, allowing the fuzzing tool to detect the crash through a failure to reconnect.

3) Fuzzing with Boofuzz: The Boofuzz agent creates a series of dynamically modified payloads and sends them to the target. The $fuzz\_target.py$ script is responsible for the fuzzing process and determining the input:

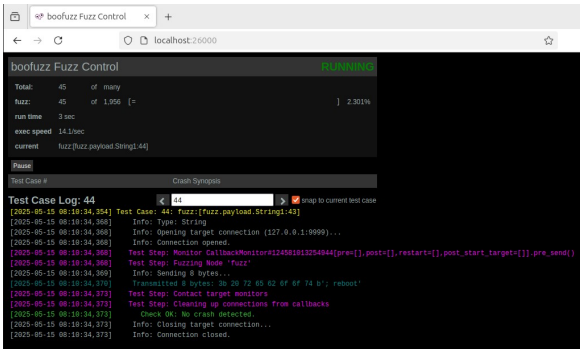

Fig. 1. Boofuzz's Web UI when running the fuzzing script.

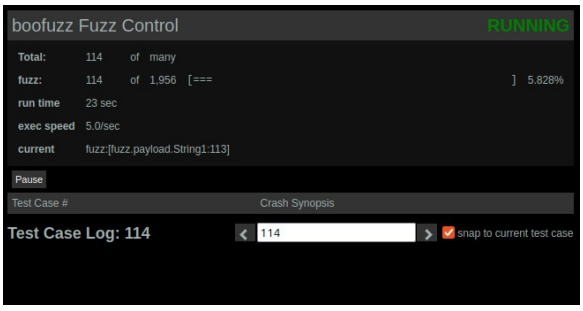

Fig. 2. Boofuzz's Web UI when the server was terminated.

```
session = Session(
    target=Target(connection =
    SocketConnection("127.0.0.1", 9999,
    proto='tcp')),
    web_port=26000,
    fuzz_loggers=[FuzzLoggerText()]
)

s_initialize("fuzz")
if s_block_start("payload"):
    s_string("HELLO", fuzzable=True)
s_block_end()

session.connect(s_get("fuzz"))
session.fuzz()
```

The agent uses a fuzzable string that gets modified for each test in an attempt to trigger unusual states on the server.

The Figure 1 illustrates the Boofuzz web interface when running the fuzzing script. Through the built-in web interface (available on port 26000), real-time monitoring of active tests, process progress, and target connection status is provided. As shown in Figure 2, the server was terminated during the test, suggesting that a particular combination of input (fuzzed payload) caused an error. Recording this failure via the interface helps to confirm that the fuzzing tool successfully detected the crash.

4) Fuzzing Log Analysis: After the fuzzing process is complete, a $fuzz\_log.csv$ file is created containing the information of each test case. The $analyze\_fuzz\_log.py$ script analyzes this file and determines if there was a crash and what the final payload was:

```
(fuzzenv) admin@Ubuntu:~/Documents/fuzz$ python3 analyze_fuzz_log_v2.py
✅ Total test cases executed: 114

🚨 Target likely crashed after this input:
  Test Case: 114
  Payload: b'CCCCCC'
```

Fig. 3. Fuzzing summary.

```
with open(log_path, encoding="utf-8")
    as f:
    lines = f.readlines()
...
if "Cannot connect to target" in line:
    crash_detected = True
    break
```

The outcome, shown in Figure 3, summarizes how many tests were performed and which input caused the server to crash. The implemented scripts can be found here.

5) Results Observations: The fuzzing process was successfully completed using Boofuzz, with a simple TCP socket server being targeted. A total of 114 test cases were created during execution, before the server crashed, each containing slightly modified versions of the original input message. Analysis of the log file ($fuzz\_log.csv$) using the $analyse\_fuzz\_log.py$ tool revealed that the server terminated abruptly during the 114^{th} test when it received the character sequence $'CCCCCC'$ as input. This sequence had been intentionally set on the server as a secret shutdown stimulus to serve as an artificial crash for evaluating the effectiveness of fuzzing. The successful detection of this input confirms that Boofuzz:

1) can generate payloads that reveal vulnerabilities (or, in this case, predefined crash conditions)
2) it recognizes connection failure as an indication of a potential target crash (e.g., a server crash)
3) tt provides an easy-to-use interface (web UI) to monitor the process and facilitate troubleshooting.

The Boofuzz interface clearly showed the termination point and the exact test case that preceded the connection failure. This information was reinforced by the log file analysis, which is a simple yet essential tool for understanding the target's behavior under different input conditions. For example, combining Boofuzz tools with Python-based post-analysis scripts provides an efficient yet lightweight approach to the initial fuzz testing of network applications, even in environments with limited computing power, such as a basic virtual machine (VM). Although this approach is based on an artificial and fully controlled crash condition, it highlights the basic logic and capability of fuzzing tools to induce unexpected situations through automated input generation. In real vulnerability scenarios, however, the inputs leading to crashes may be much more complex. This requires more careful protocol modeling (stateful fuzzing), the use of advanced heuristics, and even the monitoring of the target's behavior at the memory level (e.g., via instrumentation). Furthermore, relying on the 'crash = no response' hypothesis may result in false positives, particularly when the target experiences delays or temporary loss of connectivity. Future work could involve extending the scenario

by using fuzzers that provide deeper introspection, such as AFL++ or libFuzzer at the binary level, or incorporating honeypot mechanisms to monitor and record the server's exact behavior during an attack. Additionally, introducing an actual bug, such as a parsing or buffer overflow error, combined with fuzzing and analysis tools like GDB, would enable a more thorough evaluation of fuzzing's effectiveness in identifying safety-critical bugs.

V. Conclusion

A. What we learned from analyzing Zero-Day Exploits and Fuzzing

Dealing with zero-day attacks and the fuzzing process has highlighted both the challenges involved and the importance of proactively detecting vulnerabilities before they can be exploited. The study of zero-day cases revealed that their lack of a signature or known pattern makes them extremely difficult to detect using traditional methods. Such attacks often succeed because they exploit unknown bugs in the code that remain invisible to conventional static or manual testing tools. The practical application of fuzzing has demonstrated its potential as an automated detection tool for such bugs. Even in a simplified environment such as the one implemented in this study, it was demonstrated that systematic input generation can lead to unpredictable situations, such as a server crash. As fuzzing works without prior knowledge of vulnerabilities, it is directly related to the concept of zero-day detection. In general, this study has emphasized the importance of fuzzing as a tool to prevent zero-day attacks before they can be exploited by malicious actors. However, it also revealed the challenge of fully covering all possible entry scenarios, underscoring the need for continuous research and improvement of these techniques.

B. How useful are Fuzzers in practice

Fuzzers have proven particularly effective in automatically detecting bugs and vulnerabilities in software and systems. Unlike traditional testing methods, which often rely on manual analysis or limited test cases, fuzzing involves the mass and random generation of inputs, thus identifying cases that would otherwise be difficult to detect. Many large software companies, such as Google, Microsoft, and Mozilla, use extensive fuzzers in the development and quality control stages precisely because they can identify critical bugs that would otherwise remain dormant until they are exploited. In environments where security is critical, such as browsers, operating systems, and applications that process untrusted input, fuzzers have led to the disclosure of many zero-day vulnerabilities. Even in educational or experimental environments, such as the one described in this paper, using a tool like Boofuzz immediately highlights the advantages of fuzzing: automation, speed, and the ability to collect data for analysis. Therefore, the practical application of fuzzers is essential for a comprehensive framework for safe software development.

C. Suggestions for Further Study

Analyzing zero-day vulnerabilities and using fuzzing are important starting points for a deeper investigation of advanced vulnerability identification and prevention techniques. Future studies can be extended in several directions.

- Linking to machine learning (ML) techniques: the use of ML technologies can enhance the effectiveness of fuzzers through 'intelligent' input selection or identification of suspicious target behavior. Research efforts are underway to use supervised or reinforcement learning to guide fuzzing more specifically towards 'interesting states' of the program.
- Combination with Honeypots: Integrating honeypots into a system can provide valuable information on zero-day attacks in a real environment. This information can be used to train ML models or update fuzzing strategies.
- Studying fuzzing in other attack surfaces: beyond applications based on sockets or simple protocols, the use of fuzzing in more complex environments is worth exploring, such as APIs, web applications, GUI-based programs, and even device firmware.
- Crash report analysis and automatic categorization: Once a crash has been found, the next challenge is to quickly analyze and evaluate its severity. Tools that offer crash deduplication or automatic categorization can further increase the productivity of fuzzing.

In general, this work could form the basis for more advanced projects that combine traditional security techniques with modern analysis and automation methods.

References

[1] L. Bilge and T. Dumitraş, "Before we knew it: an empirical study of zero-day attacks in the real world," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 833–844. [Online]. Available: https://doi.org/10.1145/2382196.2382284

[2] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense*, ser. LSAD '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 131–138. [Online]. Available: https://doi.org/10.1145/1162666.1162671

[3] L. Ablon and A. Bogart, *Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits*. Santa Monica, CA: RAND Corporation, 2017.

[4] D. Kushner, "The real story of stuxnet," *IEEE Spectrum*, vol. 50, no. 3, pp. 48–53, 2013.

[5] D. Goodin, "Nsa-leaking shadow brokers just dumped its most damaging release yet," *Ars Technica*, 2017.

[6] N. Imtiaz, B. Murphy, and L. Williams, "How do developers act on static analysis alerts? an empirical study of coverity usage," 08 2019.

[7] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[8] S. Khatiwada, M. Tushev, and A. Mahmoud, "Just enough semantics: An information theoretic approach for ir-based software bug localization," *Information and Software Technology*, vol. 93, pp. 45–57, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916302269

[9] C. Poncelet, K. Sagonas, and N. Tsiftes, "So many fuzzers, so little time: Experience from evaluating fuzzers on the contiking network (hay)stack," ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556946

[10] J. Neystadt, "Automated penetration testing with white-box fuzzing," *Microsoft*, 2008. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10)?redirectedfrom=MSDN
[11] K. Ren, W. Qiang, Y. Wu, Y. Zhou, D. Zou, and H. Jin, "An empirical study on the effects of obfuscation on static machine learning-based malicious javascript detectors," 07 2023, pp. 1420–1432.
[12] Blue Goat Cyber. (2024) Unlocking runtime insights: The essential guide to dynamic code analysis. Accessed: 2025-05-15. [Online]. Available: https://bluegoatcyber.com/blog/unlocking-runtime-insights-the-essential-guide-to-dynamic-code-analysis/
[13] OnlineHashCrack. (2025) Cuckoo sandbox malware analysis tutorial. Accessed: 2025-05-17. [Online]. Available: https://www.onlinehashcrack.com/guides/security-tools/cuckoo-sandbox-malware-analysis-tutorial.php
[14] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision." 01 2005, pp. 17–30.
[15] Fortinet. (2025) What is heuristic analysis? detection and removal methods. Accessed: 2025-05-17. [Online]. Available: https://www.fortinet.com/resources/cyberglossary/heuristic-analysis
[16] Corelight, "Anomaly-based detection: Machine learning for threat detection," 2025, accessed: 2025-05-17. [Online]. Available: https://corelight.com/resources/glossary/anomaly-based-detection
[17] V. L. Le, I. Welch, X. Gao, and P. Komisarczuk, "Anatomy of drive-by download attack," vol. 138, 01 2013, pp. 49–58.
[18] J. Osamor, J. Odum, C. Iwendi, F. Olajide, I. Peter-Osamor, V. Onyenagubom, and I. Ayodele, "Ethical implications of wannacry: A cybersecurity dilemma," *International Conference on Cyber Warfare and Security*, vol. 20, pp. 354–360, 03 2025.
[19] Wikipedia contributors, "Zerodium — Wikipedia, the free encyclopedia," 2025, [Online; accessed 17-May-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Zerodium&oldid=1287769712
[20] L. H. Newman, "A top-shelf iphone hack now goes for $1.5 million," *WIRED*, 2016. [Online]. Available: https://www.wired.com/2016/09/top-shelf-iphone-hack-now-goes-1-5-million/
[21] Startup Defense, "Exploit kit fundamentals: A complete security guide," 2023, accessed: 2025-05-17. [Online]. Available: https://www.startupdefense.io/cyberattacks/exploit-kit
[22] CrowdStrike, "How ransomware spreads: Top 10 infection methods," 2023, accessed: 2025-05-17. [Online]. Available: https://www.crowdstrike.com/en-us/cybersecurity-101/ransomware/how-ransomware-spreads/
[23] CYFIRMA, "Malware detection evasion techniques," 2023, accessed: 2025-05-17. [Online]. Available: https://www.cyfirma.com/research/malware-detection-evasion-techniques/
[24] Wikipedia contributors, "Sandworm (hacker group) — Wikipedia, the free encyclopedia," 2025, [Online; accessed 18-May-2025]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Sandworm_(hacker_group)&oldid=1286848691
[25] A. Shaikh and P. Gupta, *Advanced Signature-Based Intrusion Detection System*, 01 2023, pp. 305–321.
[26] Corelight, "What is signature-based detection?" 2023, accessed: 2025-05-17. [Online]. Available: https://corelight.com/resources/glossary/signature-based-detection
[27] Fortinet, "What is sandboxing? sandbox security and environment," 2023, accessed: 2025-05-17. [Online]. Available: https://www.fortinet.com/resources/cyberglossary/what-is-sandboxing
[28] C. Wagner, A. Dulaunoy, G. Wagener, and A. Iklody, "Misp: The design and implementation of a collaborative threat intelligence sharing platform," in *Proceedings of the 2016 ACM Workshop on Information Sharing and Collaborative Security (WISCS)*. ACM, 2016, pp. 49–56. [Online]. Available: https://dl.acm.org/doi/10.1145/2994539.2994542
[29] AlienVault, "Open threat exchange (otx)," https://otx.alienvault.com/, 2023, accessed: 2025-05-18.
[30] OWASP Foundation, "Virtual patching best practices," 2023, accessed: 2025-05-18. [Online]. Available: https://owasp.org/www-community/Virtual_Patching_Best_Practices
[31] Author(s), "Advancing cybersecurity with honeypots and deception strategies," *Information*, vol. 12, no. 1, p. 14, 2021.