# Zero-Day Vulnerabilities and Fuzzing:

## From Threat Analysis to Exploit Discovery



Internet security, Postgraduate Studies Program
Communication Networks and Systems Security

Maria Nefeli Ntantouri
AEM: 180

# TABLE OF CONTENTS

# 01.

## Introduction

# Introduction to Zero-Day Vulnerabilities

### What are they?

- Security flaws unknown to the software vendor
- No official patch or fix available

### Why Are They Important

- High exploitation potential
- Used in targeted attacks, cyber espionage, cyber warfare

### Purpose of this Work

- Explore zero-day discovery, exploitation, and defense methods
- Present fuzzing as a proactive detection technique
- Demonstrate fuzzing in action with BooFuzz

# 02.

# Theoretical Background

# The Lifecycle of a Zero-Day Vulnerability

- Discovery
- Weaponization
- Delivery & Exploitation
- Persistence & Evasion
- Impact
- Disclosure or Patch

- Exploited before the vendor is aware
- Often discovered by hackers, researchers, or agencies
- Sold in dark markets or used for espionage
- Lifecycle ends when patch or mitigation is applied

# Static vs. Dynamic Analysis

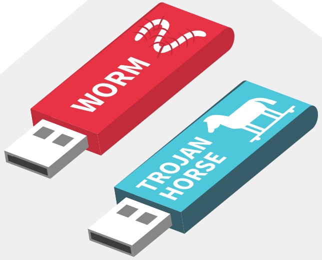
## Static Analysis

- Analyzes code without execution
- Detects vulnerabilities via code inspection
- Fast and scalable
- May produce false positives
- Struggles with obfuscated/packed code

## Dynamic Analysis

- Observes behavior during execution
- Captures runtime behavior, system interactions
- Slower, but context-aware
- Better at catching real-world issues
- Can bypass obfuscation through execution tracing

Combine **both** for comprehensive vulnerability discovery.



# Heuristics & Pattern Matching

## Key Characteristics:

- Signature-based detection: Relies on databases of known threat indicators.
- Rule-based logic: Matches sequences, structures, or suspicious API usage.
- Fast & low-resource: Suitable for real-time scanning.
- Limited to known threats: Ineffective against novel zero-days or polymorphic malware.

## Example Tools

**Snort, ClamAV, antivirus engines**

## Definition

Heuristics and pattern matching involve detecting vulnerabilities or malicious behavior based on known patterns, signatures, or suspicious constructs commonly found in harmful code.

# ML-Based Detection

## What is it?

ML-based detection uses algorithms to learn and recognize patterns of normal vs. abnormal behavior to identify previously unseen threats.

## Applications

Anomaly-based intrusion detection
Malware classification
Behavioral fingerprinting

## Techniques

- **Supervised Learning**: Trained on labeled datasets
- **Unsupervised Learning**: Detects anomalies without prior labels
- **Reinforcement Learning**: Learns through interaction and feedback

## Challenges

- High false positive rates
- Need for quality training data
- Model interpretability

# Acquisition & Weaponization

## Acquiring the Vulnerability

Discovered by:

- Security researchers
- Hacktivists / Cybercriminals
- Government agencies

Sold in:

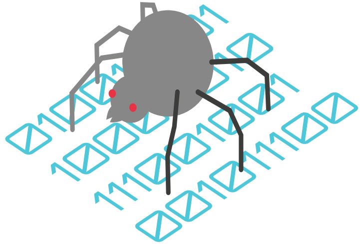
- Dark Web markets (e.g., Zerodium, Exodus)
- Bug bounty programs

## Weaponization

- Exploit code is developed to trigger the vulnerability
- Delivered via:
    - Custom malware
    - Exploit kits (e.g., RIG, Neutrino)
- Often tested in controlled environments before use

## Real-World Impact

**High-value targets: iOS, Windows, SCADA systems**



# Delivery, Persistence, and Impact

## Delivery Methods

- Phishing (malicious links/attachments)
- Drive-by downloads (compromised websites)
- Malvertising (infected ads)

## Avoidance/Evasion Techniques

- Code obfuscation & encryption
- Polymorphism (changes form on each run)
- Anti-sandbox logic (delays or avoids execution in virtual environments)

## Persistence Techniques

- Scheduled tasks, registry edits
- Installing rootkits
- Auto-start services

## Consequences

- Data theft or destruction
- Espionage (government or corporate)
- Disruption of critical infrastructure
- Long-term undetected presence in systems

# Detection & Containment Approaches

## Signature-Based Detection

- Relies on known patterns (signatures)
- Fast and efficient for known threats
- Limitation: Ineffective for zero-days

## Behavior-Based Detection

- Monitors deviations from normal behavior
- Uses ML, heuristics, anomaly detection
- Effective for unknown or evolving threats

## Sandboxing

- Isolated environment for testing suspicious files
- Captures system calls, network activity, file/registry changes
- Tools: Cuckoo Sandbox, FireEye, custom VMs

# 03.

# Fuzzing: Theory and Practice

# Fuzzing Tools


## Goal

Trigger crashes, memory errors, or unexpected behavior → Reveal unknown vulnerabilities (incl. zero-days).

## Why It Matters

1. Scalable and repeatable
2. Effective on binary and source code
3. Especially useful for edge cases & logic flaws

## Definition

Fuzzing is an automated testing technique that sends random, malformed, or unexpected inputs to software.

## Types

- Black-box (no internal knowledge)
- White-box (uses source code)
- Grey-box (partial knowledge, e.g. with instrumentation)

# 04.

# Fuzzing Scenario

# Demo Setup

```python
import socket

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)
        print(f"[+] Listening on {HOST}:{PORT}")

        while True:
            conn, addr = s.accept()
            with conn:
                print(f"[+] Connection from {addr}")
                while True:
                    try:
                        data = conn.recv(1024)
                        if not data:
                            break
                        print(f"[>] Received: {data}")
                        if SECRET_SEQUENCE in data:
                            print("[!] Secret sequence received. Shutting down.")
                            return  # Τερματίζει όλο το server
                    except ConnectionResetError:
                        print("[!] Connection reset by peer.")
                        break

if __name__ == "__main__":
    start_server()
```

*vuln_server.py*

## Fuzzing Environment

- Ubuntu VM (VirtualBox)
- Python 3 in virtual environment
- Tool: Boofuzz (open-source fuzzing library)

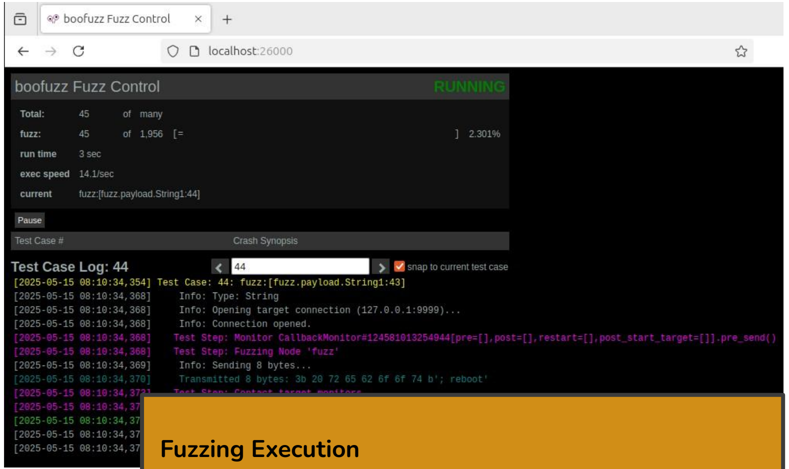
## Fuzzing Scenario

- **Goal**: Identify crash conditions in a vulnerable TCP server
- **Target**: Python-based socket server on port 9999
- **Crash Trigger**: Receives special input b'CCCCCC'
- **Crash**: Server terminates via return, simulating application crash
- **Detection**: Crash inferred via connection loss

# Fuzzing in Action

```python
from boofuzz import *

def main():
    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()

if __name__ == "__main__":
    main()
```

*fuzz_target.py*




```
(fuzzenv) admin@Ubuntu:~/Documents/fuzz$ python3 vuln_server.py
[+] Listening on 127.0.0.1:9999
[+] Connection from ('127.0.0.1', 54938)
[>] Received: b'!@#$%%^#$%#$@#$%$$@#$%^^**(()'
[+] Connection from ('127.0.0.1', 54946)
[+] Connection from ('127.0.0.1', 54960)
[>] Received: b'$(reboot)'
[+] Connection from ('127.0.0.1', 54966)
[>] Received: b'$;reboot'
[+] Connection from ('127.0.0.1', 54972)
[>] Received: b'%00'
[+] Connection from ('127.0.0.1', 54976)
[>] Received: b'%00/'
[+] Connection from ('127.0.0.1', 54988)
[>] Received: b'%01%02%03%04%0a%0d%0aADSF'
```

```
(fuzzenv) admin@Ubuntu:~/Documents/fuzz$ python3 fuzz_target.py
/home/admin/fuzzenv/lib/python3.12/site-packages/boofuzz/connections/socket_connection.py:79
ing: SocketConnection is deprecated and will be removed in a future version of Boofuzz. Use
derived from BaseSocketConnection instead.
  warnings.warn(
[2025-05-19 11:31:09,335]     Info: Web interface can be found at http://localhost:26000
[2025-05-19 11:31:09,357] Test Case: 1: fuzz:[fuzz.payload.String1:0]
[2025-05-19 11:31:09,357]     Info: Type: String
[2025-05-19 11:31:09,357]     Info: Opening target connection (127.0.0.1:9999)...
[2025-05-19 11:31:09,358]     Info: Connection opened.
[2025-05-19 11:31:09,361]   Test Step: Monitor CallbackMonitor#124245000222288[pre=[],post=[
,post_start_target=[]].pre_send()
[2025-05-19 11:31:09,361]   Test Step: Fuzzing Node 'fuzz'
[2025-05-19 11:31:09,361]     Info: Sending 29 bytes...
[2025-05-19 11:31:09,362]     Transmitted 29 bytes: 21 40 23 24 25 25 5e 23 24 25 23 24 40 23 24 25 24 2
4 40 23 24 25 5e 5e 2a 2a 28 28 29 b'!@#$%%^#$%#$@#$%$$@#$%^^**(()'
```

**Fuzzing Execution**

- Script: fuzz_target.py
- Boofuzz agent sends modified inputs to server

**Monitoring via Web UI**

- Boofuzz web interface on **localhost:26000**
- Real-time test tracking

# Results

**Crash Identified**

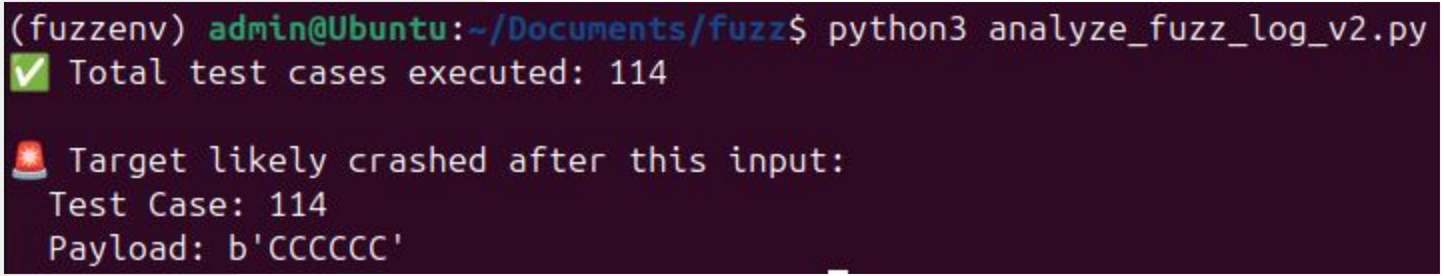
- Input number: 114
- Payload: b'CCCCCC'
- Result: Server crashed and terminated execution

**Fuzz Log Analysis**

- fuzz_log.csv recorded all test cases
- analyze_fuzz_log.py script parsed final payload

## Conclusion

- Payload b'CCCCCC' is the trigger
- Confirms success of fuzzing setup & Boofuzz detection



# 05.

# Conclusion

# Conclusions & Future Outlook

| Key Learnings | Role of Fuzzing | Future Perspectives |
|---|---|---|
| Zero-day vulnerabilities are dangerous due to their stealth and impact | Powerful for discovering unknown bugs and crash conditions | Integration of machine learning to improve fuzzing coverage and analysis |
| Their life cycle includes discovery, weaponization, delivery, and persistence | Helps defenders simulate attacker behavior before the attacker does | Use of honeypots to detect exploitation attempts and gather threat intelligence |
| Defense is complex: a mix of detection, mitigation, and proactive techniques is needed | Especially useful for uncovering input-based vulnerabilities | Combining sandboxing & threat sharing platforms for faster, coordinated defense |


# THANKS

## DO YOU HAVE ANY QUESTIONS?