WEAPONIZING VULNERABILITIES: THE ART OF EXPLOIT DEVELOPMENT

GROUP 2

Course: Advanced Cyberwarfare Programme

Course Code: ACW903: Nation-State Cyber Operations

Instructor Name: Aminu Idris

Date: Sep 24, 2025

Version 1.0

Introduction

Exploit development is the art of turning a software bug into a functional attack. It typically starts with fuzzing – automated testing that feeds random or malformed inputs to a program to uncover crashes. Fuzzers like AFL++ or libFuzzer instrument the code and iteratively mutate inputs, aiming to cover new code paths. Once a crash is found, analysts reverse-engineer the fault to understand the vulnerability (e.g. a buffer overflow or format string). They then write a proof-of-concept (PoC) exploit – often in Python or C – that reliably triggers the flaw. Finally, they craft payloads and use advanced techniques (e.g. Return-Oriented Programming) to bypass mitigations like DEP/NX (non-executable memory) and ASLR (address randomization). This paper details each step: fuzzing and discovery, PoC writing, payload generation, and defeating modern defenses, with examples from real-world exploits. By the end, readers will understand how complex exploits are engineered and how defenders can counter them.

Key Techniques & Methodologies

Fuzzing for Vulnerability Discovery:

Fuzzing automates the discovery of bugs by throwing unexpected or random inputs at software . Modern coverage-guided fuzzers like AFL++ use compile-time instrumentation and genetic algorithms to intelligently mutate inputs and find crashes . For example, AFL++ can compile a vulnerable program and then mutate seed inputs to trigger a buffer overflow, while measuring code coverage to guide future inputs. Fuzzing excels at revealing memory-safety flaws (stack/heap overflows, use-after-free, integer bugs) and protocol or file-format parsing errors . In practice, pentesters collect valid input examples, set up the program under the fuzzer, and let it

run for hours or days until unique crashes appear . Each crash is then triaged: developers examine it with debuggers (e.g. GDB, WinDbg) to confirm the bug.

Reproducing and Analyzing Crashes:

Once a crash is found, the next step is to reproduce it deterministically and analyze it in a debugger. A common method is to rebuild the target with debug symbols and use tools like cyclic from Pwntools to generate a patterned input that pinpoints the exact offset of the overwrite . The developer sends the crash-inducing input to the program under a debugger to see which instruction or address was overwritten. For example, a cyclic buffer might reveal that the instruction pointer was set to 0x61616167, indicating an offset of 24 bytes . This offset tells the exploit writer how many bytes of padding are needed before injecting a crafted value. At this stage, simple PoCs often just crash the program (e.g. by overwriting a return address with an invalid value), demonstrating the vulnerability exists.

Writing Proof-of-Concept Exploits:

With the vulnerability understood, the next step is to write code that triggers it with a malicious payload. In modern exploit development, Python (often with the Pwntools library) is a popular choice for PoC exploits. Pwntools provides convenient functions for process/network interaction, payload generation, and common exploit patterns. A typical exploit script will (1) set up a process or network connection, (2) compute the exact offset and gadget addresses, and (3) build the payload. For example, a Python exploit might look like:

```
from pwn import *

context(os='linux', arch='amd64')

# Load binary and/or remote target

binary = ELF('./vulnerable_app')

p = process(binary.path)

# Determine offset (found via cyclic or analysis)

offset = 120

# Craft payload: padding + new return address + filler

payload = b"A" * offset + p64(0xdeadbeef) + b"\n"
```

```
p.sendline(payload)

p.interactive()
```

This snippet shows a simple stack-based overflow after offset bytes of padding, we overwrite the saved return address with 0xdeadbeef. In a real exploit, instead of a dummy address, one would place the address of a gadget or function (e.g. system) and the required arguments. Pwntools also provides cyclic_find() to compute offsets, and shellcraft to generate shellcode or ROP chains . By combining these tools, exploit writers quickly iterate: send a payload, check if it gives a shell or desired effect, then refine.

Payload Construction – Shellcode and ROP:

Modern OS defenses typically mark data regions (stack, heap) as non-executable (DEP/NX). To bypass this, exploit writers avoid injecting raw code on the stack. Two common techniques are return-to-libc and ROP (Return-Oriented Programming). In ret2libc, the exploit redirects execution to existing library functions. For instance, if the attacker can overflow the return address, they may set it to system() in libc and place "/bin/sh" address on the stack . This calls system("/bin/sh") without injecting new code. But if ASLR is enabled, the libc base address is randomized. Polito's writeup notes that "if ASLR is enabled … the attacker must leak a pointer … to calculate the base address" . A format-string bug or heap leak might provide that pointer, allowing calculation of system()'s actual address. Once known, the payload is built by packing the addresses appropriately, e.g. using p64() for 64-bit values.

If a single system() call is insufficient (e.g. no convenient string), or multiple steps are needed, attackers use ROP. Wikipedia defines ROP as a method "that allows an attacker to execute code in the presence of defenses such as executable-space protection" . Instead of jumping to the system, the attacker chains together small instruction sequences ("gadgets") that end in a ret. For example, gadgets to set up registers and call VirtualAlloc can disable DEP (see below). The exploit stack frame becomes a sequence like: [addr_of_gadget1][addr_of_gadget2][…], where each ret jumps to the next gadget. ROP defies DEP because it reuses existing code that is already executable . Building a ROP chain is laborious: one must find suitable gadgets (using tools like ROPgadget or assembly analysis) and order them to perform the desired function. As NCC Group explains, on Windows a common ROP strategy is to call APIs like VirtualProtect or VirtualAlloc to mark the stack executable, and then jump to injected shellcode . This kind of chain is constructed by writing gadget addresses into the overflowed return address and stack, often with Python scripting.

Bypassing ASLR (Address Randomization):

ASLR randomizes memory segments each run (stack, heap, libraries) to foil static addresses. A strong ASLR means the attacker cannot know addresses of code or memory a priori. However, 32-bit ASLR is limited. Research notes that 32-bit Linux randomizes only 16 bits of the address, making brute-force feasible . Even on 64-bit systems, a single leaked pointer can defeat ASLR. For example, if a vulnerability allows reading a function pointer from memory, that leak exposes the random base of a module. Exploit developers may use format-string or info-leak bugs for this. Once the base is known, offsets to gadgets or functions are fixed, allowing ROP or ret2libc. In practice, if an attacker can leak any address (heap metadata, stack canary, libc symbol), they recalculate offsets to all needed code, effectively negating ASLR .

Evasion of Other Protections:

Aside from ASLR/DEP, modern binaries have other defenses. Stack canaries (random values placed before the return address) will abort on overflow. Exploit code can try to overwrite data beyond the canary without corrupting it, or leak the canary first. SafeSEH (in Windows) or SEHOP can block SEH overwrites , so exploitation may use stack pivoting or avoid SEH entirely. RELRO makes the Global Offset Table read-only to stop GOT overwrites. Attackers adapt by using ROP on the main executable or modules that lack PIE (position-independent executables). Code signing and Control-Flow Guard (Windows) also raise the bar. In general, exploit authors either find a way around each layer or chain multiple vulnerabilities together. For example, one sub-vulnerability might leak a stack cookie, while another buffer overflow actually hijacks execution.

Workflow Tools and Automation:

Throughout exploit development, tools aid each phase. Fuzzers (AFL, Honggfuzz, Peach) find bugs; disassemblers/decompilers (IDA, Ghidra) help analyze the code; debuggers (gdb, pwndbg, WinDbg) let authors step through execution and test payloads. Scripts in Python or C automate payload delivery. Frameworks like Metasploit can assist in turning a PoC into a payload for known services. The modern workflow is iterative: fuzz → analyze → prototype exploit → adjust bypasses → test→ refine. The entire process relies on deep system knowledge (calling conventions, memory layout, architecture specifics) and creativity. As one security blog emphasizes, fuzzing is “often the first step to develop your zero-day exploit” , but taking the crash to a working shell is a complex art.

Real-World Case Study: A Technical Analysis of EternalBlue

The EternalBlue exploit is a prime example of a highly sophisticated, real-world attack that leveraged a critical vulnerability to achieve global, devastating impact. EternalBlue targets a

remote code execution vulnerability (MS17-010, also known as CVE-2017-0144) in Microsoft's Server Message Block (SMB) version 1 protocol. SMB is a network file-sharing protocol used by Windows machines, and the vulnerability exists because the SMBv1 server mishandles specially crafted packets, allowing remote attackers to execute code on a target computer.

The exploit is a testament to the fact that modern exploits are often not a single bug but a complex chain of multiple vulnerabilities. EternalBlue leveraged a sequence of three distinct bugs to achieve its objective. The first two bugs, a "wrong casting bug" and a "wrong parsing function bug," were used to cause a buffer overflow. A third bug, a "non-paged pool allocation bug," was then used to manipulate the heap's memory layout to make the exploit reliable. The existence of multiple CVEs associated with the MS17-010 bulletin (e.g., CVE-2017-0143 to CVE-2017-0148) further underscores this multi-bug chain. This demonstrates that a single, catastrophic exploit often represents the culmination of a deep understanding of multiple software components and their interactions, allowing an attacker to chain together seemingly minor flaws for a devastating effect.

The EternalBlue exploit is a multi-stage process that meticulously orchestrates memory corruption to gain control of a target system. The exploit begins with an initial SMB handshake to establish a connection. The attacker then sends a large amount of File Extended Attributes (FEA) data to fill the heap, a technique known as heap spraying. This heap spraying is crucial for increasing the chances that a srvnet struct, containing the attacker's shellcode, will be placed at a predictable memory location. The shellcode used by EternalBlue was a backdoor known as DoublePulsar, which allowed for a more persistent and controllable compromise.

Next, the exploit leverages the non-paged pool allocation bug (Bug 3) to create a "buggy chunk" in the heap. This action makes room for the NTFeaList to be stored. The final part of the FEA data is then sent, which triggers the "wrong casting bug" and "wrong parsing function bug" to cause a buffer overflow. This overflow corrupts a srvnet struct located adjacent to the freed chunk, overwriting its handler function's return address with a pointer to the injected DoublePulsar shellcode. The final step in the exploit chain is the disconnection of the SMB session. As the connection closes, the handler function for each srvnet struct is executed, and the handler for the overflowed struct executes the attacker's shellcode. This methodical process demonstrates the intricate engineering required to bypass memory protections and achieve remote code execution.

The full, devastating impact of EternalBlue was realized when it was used in the WannaCry and NotPetya attacks. These events transformed the exploit from a mere vulnerability into a global catastrophe. The exploit was part of a trove of cyber tools stolen from the U.S. National Security Agency (NSA) by a hacking group known as the Shadow Brokers. The exploit was publicly leaked on April 14, 2017, and less than a month later, WannaCry began its rampage.

WannaCry was a ransomware worm that encrypted data on infected systems and demanded a ransom payment in Bitcoin. It leveraged EternalBlue to spread automatically across networks by scanning for vulnerable systems with an open SMB port. This wormable nature was a major paradigm shift, allowing a single infection to compromise an entire corporate network. The attack infected over 250,000 systems across 150 countries, causing an estimated $4 billion in damages.

NotPetya, which appeared a month later, was even more destructive. Disguised as ransomware, it was actually a "wiper" designed for pure destruction, permanently encrypting the master boot record and master file table of infected machines. NotPetya's global spread happened because multinational corporations with Ukrainian offices became infection vectors for their worldwide networks. The malware used the EternalBlue exploit to spread rapidly, but it also incorporated other tools like Mimikatz to steal credentials and trusted administrative tools like PsExec to move laterally across the network. This combination of a wormable exploit with post-exploitation tools allowed it to turn a single infected system into a launch point for an enterprise-wide disaster, causing over $10 billion in damages. The true legacy of EternalBlue is its role in the evolution of cyber threats from isolated compromises to enterprise-wide devastation.

EternalBlue Attack Timeline and Impact

Date	Event	Significance
Before 2017	The NSA develops and uses the exploit for years	Demonstrates the power of offensive cyber tools developed by state actors
March 14, 2017	Microsoft releases security bulletin MS17-010	A patch is available to fix the vulnerability, two months before the major attacks begin
April 14, 2017	The Shadow Brokers leak EternalBlue to the public	The exploit is now in the public domain, lowering the barrier to entry for attackers

May 12, 2017	The WannaCry ransomware attack begins	First major use of a wormable exploit for a global attack, causing billions in damages
May 13, 2017	The WannaCry "kill switch" is discovered	An operational flaw in the malware halts its spread, preventing further encryption and damage
June 27, 2017	The NotPetya "wiper" attack begins	A more destructive attack demonstrates the lateral movement capabilities enabled by the exploit

CVE-2021-3156 “Baron Samedit” (sudo Heap Overflow): On the Linux side, CVE-2021-3156 is a recent example where fuzzing turned into root. Researchers at Qualys used AFL++ to fuzz the sudo utility and found a heap-based buffer overflow that had existed for nearly a decade . The bug occurs in argument parsing (the shell escaping code); by providing specially crafted arguments, an attacker could overflow a heap buffer. Qualys confirmed that “successful exploitation allows any unprivileged user to gain root privileges” on vulnerable Unix-like systems . They released proof-of-concept exploits (in C and Python) that simply gain a root shell. This case shows the full cycle: AFL++ discovered the flaw , researchers debugged it, and then wrote exploits to demonstrate it on Ubuntu, Debian, etc. . Notably, CVE-2021-3156 had nothing to do with modern mitigations – it was a logic bug – but it underscores that even mature, widely-used tools can harbor critical overflows. The swift PoCs validated the severity and guided distro updates.

These examples illustrate the exploit process: fuzzing uncovered the bug, analysis revealed how to hijack control flow, and payload code was crafted to execute shellcode despite DEP/ASLR. EternalBlue’s chain involved leaking an address and using a complex ROP-style gadget , while Baron Samedit’s heap overflow could be triggered with a single malicious command line. Both required deep debugging and custom exploit code. In practice, exploit developers often share their methods (CVE write-ups, blog posts) to teach these steps to others.

Defensive Countermeasures

Organizations can defend against such exploit chains through defense-in-depth. First, system hardening is key: keep all software and libraries fully patched so known bugs are fixed. Imperva stresses that patching quickly is critical once a vulnerability is found, to prevent it from being

exploited . Developers should write safer code and consider using memory-safe languages (e.g. Rust, Go, or managed languages) for new projects. As Imperva notes, languages like C# and Java include built-in bounds checks that reduce overflow risks .

At the runtime level, employ layered mitigations:

- Enable DEP/NX to mark stacks/heaps non-executable. DEP “designates certain areas of memory as non-executable” to prevent injected shellcode . Hardware-assisted DEP (NX bit) is now default on most platforms.

- Use ASLR to randomize memory layout on each run. Modern OSs randomize stack, heap, and libraries so attackers must leak addresses to know where to jump . Blue Goat Cyber emphasizes that bypassing ASLR “will nearly always require an information leak” . Thus, system configuration should maximize entropy (e.g. full 64-bit ASLR).

- Compile code with stack canaries and Stack Cloning protections to detect or prevent overwrites of return addresses.

- On Windows, enable SafeSEH/SEHOP and CFG (Control Flow Guard) to block many exploit patterns.

- Use Full-RelRO/PIE on Linux so that the program and libraries load at random addresses and GOT entries are read-only.

- Regularly scan with fuzzers or static analysis (e.g. oss-fuzz, sanitizers) to find defects before attackers do.

Network defenses also help: deploy firewalls and intrusion prevention systems that detect exploit signatures or anomalous behavior (e.g. unusual SMB transaction patterns). For web and network services, Web Application Firewalls can block malformed requests. Logging and monitoring can catch exploit attempts (e.g. a process spawning a shell unexpectedly). As Blue Goat suggests, defense-in-depth – combining protections – is essential. Use secure coding practices (input validation, memory-safe APIs) and regular updates as primary measures . Employ intrusion detection/EDR to spot lateral movement or unusual memory operations. In cloud or virtualized environments, use micro-segmentation and least-privilege to limit damage if an exploit succeeds.

Importantly, routine fuzzing and penetration testing by defenders can preempt some attacks. If an organization fuzz-tests its own binaries, it may find critical bugs internally. Static and dynamic analysis tools can highlight suspicious code paths. Finally, educate developers about these threats: understanding how an exploit works makes it easier to avoid coding errors (e.g. never

using strcpy on unbounded input, always checking buffer lengths, etc.). In summary, while no single defense is perfect, multiple overlapping controls significantly raise the attacker's workload . As one analyst summarizes, combine DEP, ASLR, stack cookies, and rapid patching to make successful exploitation far more difficult .

Conclusion

Exploit development is a complex, multi-step process that bridges vulnerability research and low-level programming. It typically begins with automated testing (fuzzing) to uncover bugs, followed by careful analysis and PoC coding to exploit them. Advanced attackers chain techniques like ret2libc and ROP to defeat modern defenses such as DEP/NX and ASLR . Real examples like EternalBlue and CVE-2021-3156 demonstrate this chain end-to-end: from fuzzing or discovery, to crafting payloads and bypassing mitigations . The cat-and-mouse continues: as attackers refine exploit methods, defenders must continually apply layered mitigations (memory randomization, non-executable memory, safe languages) and keep systems patched . By understanding the technical process of exploit creation, security professionals can better anticipate attack vectors and strengthen their defenses.

References

BetaFred. (n.d.). Microsoft Security Bulletin MS17-010 - Critical. Microsoft Learn.

https://learn.microsoft.com/en-us/security-updates/securitybulletins/2017/ms17-010

Burdova, C. (2025, September 16). What is EternalBlue and why is the MS17-010 exploit still

relevant? What Is EternalBlue and Why Is the MS17-010 Exploit Still Relevant?

https://www.avast.com/c-eternalblue

EternalBlue Exploit: What it is and how it works – CyberIR@MIT. (n.d.).

https://cyberir.mit.edu/site/eternalblue-exploit-what-it-and-how-it-works/

GitHub. (2024, July 29). What is fuzzing and fuzz testing? GitHub.

https://github.com/resources/articles/security/what-is-fuzz-testing

Staff, T. N. (2025, May 27). Proof of Concept (POC) exploit — ThreatNG Security - External

Attack Surface Management (EASM) - Digital Risk Protection - Security Ratings.

ThreatNG Security. https://www.threatngsecurity.com/glossary/proof-of-concept-exploit

UncleSp1d3r. (2023, April 18). Exploit Development - Introduction and techniques. UncleSp1d3r Blog. *https://unclesp1d3r.github.io/posts/2023/04/exploit-development-introduction-and-techniques/*

VulnCheck - Outpace adversaries. (2025, May 23). VulnCheck. https://www.vulncheck.com/blog/understanding-exploit-proof-of-concept

Olmstead, K. (2025, February 20). *How to become an exploit Developer*. OffSec. https://www.offsec.com/cybersecurity-roles/exploit-developer/