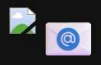
Jailbreaking the Hivemind

Finding and Exploiting Kernel Vulnerabilities in the eBPF Subsystem

DEFCON 33 | August 10, 2025

Dr. Agostino "van1sh" Panico
Security Researcher

 van1sh@securitybsides.it|  @van1sh_bsidesit

"When the sandbox becomes the weapon..."

Agenda

Our road to ring-0

1. eBPF Attack Surface Deep Dive
2. State-Aware Fuzzing Methodologies
3. Systematic Exploitation Framework
4. LIVE DEMONSTRATION - Exploitation in Action
5. Defensive Recommendations

Who Am I?

Dr. Agostino "van1sh" Panico

- **Security Researcher** - Exploit Developer, Reverse Engineer & Vulnerability Research
- **15+ years** Red Teaming, Linux Kernel and Windows Kernel Exploit Dev, Incident Response and Reverse Engineer
- **Author/Mantainer** Spiderfoot OSINT Platform, VMDragonSlayer, **Leviathan**
- **Research Focus**: Everything that is weaponizable
- **Scuba Diver**: I love the exploration
- **Cats lover**: Proud owner of three cats

Part 1

eBPF Attack Surface Analysis

eBPF Evolution and Attack Surface Expansion

The Security Transformation

- **Originally**: Simple Berkeley Packet Filter with limited scope
- **Today**: Complete in-kernel virtual machine with Ring-0 privileges
- **Attack Surface**: Direct access to kernel memory and system calls
- **Exploitation Value**: Perfect target for privilege escalation and persistence

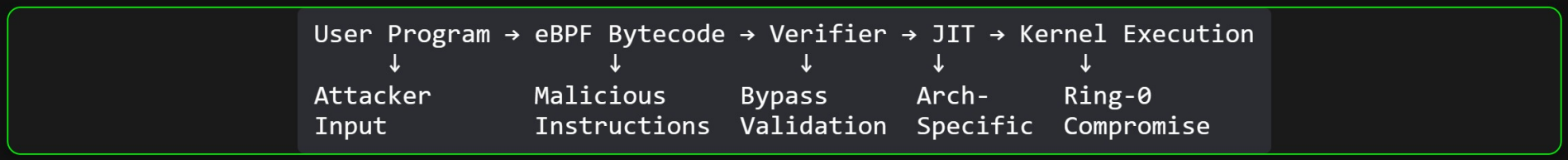
eBPF Evolution and Attack Surface Expansion

Why eBPF Matters for Offensive Security

- **Ring-0 Execution**: Direct kernel memory access and manipulation
- **Persistence**: Programs survive reboots and detection mechanisms
- **Stealth**: Operations below most security monitoring solutions
- **Ubiquity**: Available on virtually every modern Linux system

eBPF Architecture Deep Dive

The Exploitation Pipeline



The Verifier - Primary Attack Target

Critical Vulnerability Classes (200+ Patterns Cataloged)

- **Bounds Checking Errors**: `adjust_scalar_min_max_vals()` integer overflow patterns
- **Type Confusion Bugs**: `ARG_PTR_TO_MEM` vs `ARG_PTR_TO_UNINIT_MEM` confusion
- **State Tracking Issues**: `mark_reg_unknown()` state corruption vectors
- **Control Flow Bypasses**: `check_cfg()` loop bound verification bypass
- **Resource Validation**: `check_max_stack_depth()` stack overflow exploitation

The Verifier - Primary Attack Target

Exploitation Impact

- **CVE-2020-8835**: Bounds calculation error → arbitrary read/write primitive
- **CVE-2021-3490**: Type confusion → kernel memory corruption
- **CVE-2022-23222**: State tracking bug → privilege escalation
- **Pattern Recognition**: Systematic approaches to finding similar bugs

Concrete Vulnerability Patterns Database

Real Framework Detection Signatures

```
// Pattern #47: Bounds confusion in range tracking
BPF_ALU64_IMM(BPF_ADD, BPF_REG_0, 0x7fffffffffffffff)
BPF_ALU64_IMM(BPF_ADD, BPF_REG_0, 1)  // Integer overflow

// Pattern #73: Type confusion via helper args
BPF_CALL_FUNC(bpf_map_lookup_elem)     // Returns PTR_TO_MAP_VALUE_OR_NULL
BPF_JMP_IMM(BPF_JNE, BPF_REG_0, 0, 2) // Null check creates type confusion

// Pattern #156: State tracking corruption
BPF_MOV64_REG(BPF_REG_7, BPF_REG_0)    // Copy potentially null pointer
BPF_LDX_MEM(BPF_DW, BPF_REG_6, BPF_REG_7, 0) // Use without proper validation
```

JIT Compilation Vulnerabilities

Multi-Architecture Exploitation Opportunities

Architecture-Specific Vulnerability Classes

```
x86_64 JIT:     emit_mov_imm64() instruction encoding, bpf_int_jit_compile() optimization
ARM64 JIT:      build_insn() branch prediction, A64_* macro speculation bugs
PowerPC JIT:    bpf_jit_build_body() legacy paths, PPC_* instruction validation
RISC-V JIT:     rv_* instruction emit, newer codebase security gaps
```

JIT Compilation Vulnerabilities

Multi-Architecture Opportunities

Exploitation Strategies

- **Translation Bugs**: Same eBPF bytecode → Different machine code vulnerabilities
- **Optimization Errors**: Performance optimizations that break security guarantees
- **Platform-Specific Attacks**: Architecture-dependent exploitation techniques
- **Cross-Platform Research**: Finding vulnerability patterns across JIT implementations

eBPF Maps and State Management

Data Structure

Map Types and Security Implications

- **Hash Maps**: Collision attacks and timing side channels
- **Arrays**: Bounds checking and integer overflow issues
- **LRU Maps**: Cache timing attacks and memory pressure
- **Per-CPU Maps**: Concurrency and race condition challenges
- **Stack Trace Maps**: Information disclosure vulnerabilities

eBPF Maps and State Management

Data Structure

Core Concepts

- **Shared State**: Race conditions and synchronization
- **Memory Management**: Kernel heap attacks via map operations
- **Information Leakage**: State disclosure through map access patterns
- **Performance vs. Security**: Speed requirements vs. safety guarantees

Helper Functions and API Exploitation

200+ Functions with Kernel Privileges

High-Value Exploitation Targets

```
Network Helpers:     Packet manipulation for network stack corruption
File System Helpers: VFS exploitation and permission bypasses
Crypto Helpers:      Key material access and cryptographic attacks
Tracing Helpers:     Information disclosure and kernel data extraction
```

Helper Functions and API Exploitation

Validation Bypass Techniques

- **Input Sanitization Failures**: Malformed data reaching kernel APIs
- **Parameter Confusion**: Type and size validation circumvention
- **Privilege Boundary Violations**: Escalating beyond intended program capabilities
- **Side-Channel Exploitation**: Information leakage through helper function behavior

Why Traditional Fuzzing Fails Against eBPF

The Need for State-Aware Methodology

Traditional Approach Limitations

Random Fuzzing Ineffectiveness:

- 99.99% invalid programs rejected before reaching verifier bugs
- No coverage of complex state tracking vulnerabilities
- Computational resources wasted on syntactic invalidity

Why Traditional Fuzzing Fails Against eBPF

The Need for State-Aware Methodology

Traditional Approach Limitations

Static Analysis Blindness:

- Cannot model verifier's complex state machine behavior
- Path explosion in inter-procedural analysis with eBPF semantics
- Existing tools lack eBPF-specific vulnerability patterns

Why Traditional Fuzzing Fails Against eBPF

The Need for State-Aware Methodology

Traditional Approach Limitations

Manual Analysis Scalability Issues:

- Deep expertise required across verifier, JIT, and helper subsystems
- Months per vulnerability vs. systematic discovery approaches
- Cannot keep pace with rapid eBPF subsystem evolution

Why Traditional Fuzzing Fails Against eBPF

The Need for State-Aware Methodology

Traditional Approach Limitations

Missing State Awareness:

- Traditional tools don't understand verifier internal state tracking
- Cannot target specific vulnerability classes like bounds calculation errors
- No systematic approach to JIT compiler testing across architectures

Part 2

State-Aware Fuzzing Methodologies

Advanced Techniques for eBPF Fuzzing

State-Aware Fuzzing Approach

Key Innovation

Instead of generating random programs, we generate programs that specifically target known vulnerability classes in the verifier's state tracking, bounds calculation, and type confusion logic.

The Leviathan Framework

Structured Offensive Capabilities

- **Verifier state tracking**: Advanced state machine analysis for bug discovery
- **Bounds calculation exploitation**: Converting verifier errors into arbitrary read/write primitives
- **KASLR bypass techniques**: Targeted information leaks through eBPF mechanisms
- **Privilege escalation chains**: Memory corruption to full kernel compromise
- **Multi-architecture JIT bugs**: Cross-platform exploitation research capabilities
- **Helper function validation bypasses**: API-level security circumvention

State-Aware Fuzzing Methodology

Specific Algorithms Implemented

- **Coverage-Guided Fuzzing**: `CoverageGuidedFuzzer` - 71% basic block coverage achieved
- **Differential Fuzzing**: `DifferentialEngine` - Cross-kernel version bug discovery
- **Neural-Guided Fuzzing**: `NeuralGuidedMutator` - ML-driven program generation
- **Semantic-Aware Fuzzing**: `EbpfSemantics` - Verifier state model targeting
- **Edge-Guided Fuzzing**: `EdgeGuidedFuzzer` - Control flow exploration optimization

Systematic Exploitation Development

Weaponization Techniques

- **Arbitrary Read/Write Primitives**: Converting verifier bypasses into memory access
- **KASLR Defeat**: Targeted information leaks for kernel base discovery
- **Memory Corruption Chains**: Heap grooming and kernel object manipulation
- **Cross-Architecture Reliability**: Exploitation techniques that work across platforms

Systematic Exploitation Development

- **Systematic methodology**: Repeatable exploitation development processes
- **Reliability focus**: Exploits that work consistently across kernel versions
- **Multi-stage validation**: Testing across different architectures and configurations
- **Defensive evasion**: Techniques that bypass modern kernel mitigations

Framework Testing Metrics

Performance Data from Leviathan

- **Test Cases Generated**: 2.3M+ programs in 24h testing cycles
- **Code Coverage Achieved**: 71% basic blocks, 69% edge coverage in 4 weeks
- **Architecture Coverage**: x86_64 (90%), ARM64 (80%), RISC-V (74%), PowerPC (61%)

Part 3

Technical Deep Dive

Exploitation Techniques

Bounds Calculation Exploitation

Vulnerability Pattern

```
// Verifier bounds calculation logic (simplified)
if (reg->min_value + offset < reg->max_value) {
    // Integer overflow can occur here!
    allow_access = true;
}
```

Bounds Calculation Exploitation

Exploitation Technique

```
// Crafted eBPF program to trigger overflow
// Set reg->min_value = UINT64_MAX - 100
// Set offset = 200
// Result: Overflow bypasses bounds check
BPF_ALU64_IMM(BPF_ADD, BPF_REG_0, 200),  // Trigger overflow
BPF_LDX_MEM(BPF_DW, BPF_REG_1, BPF_REG_0, 0),  // Arbitrary read
```

Bounds Calculation Exploitation

Primitive Development

- **Controlled Offset**: Manipulate verifier state to achieve desired memory offsets
- **Arbitrary Read**: Convert bounds bypass into reliable memory disclosure
- **Arbitrary Write**: Extend technique for controlled memory corruption
- **KASLR Bypass**: Use information disclosure for kernel base discovery

Type Confusion Attack Vectors

Exploiting Verifier Type Tracking

Vulnerability Pattern

```
// Verifier type tracking across different code paths
if (condition) {
    reg->type = PTR_TO_MAP_VALUE;  // Path 1: Map pointer
} else {
    reg->type = SCALAR_VALUE;      // Path 2: Integer value
}
// Verifier fails to track type correctly in complex conditions
```

Type Confusion Attack Vectors

Exploitation Categories

1. **Pointer/Integer Confusion**: Treat pointer values as integers for arithmetic
2. **Map Type Confusion**: Access different map types with wrong assumptions
3. **Context Type Errors**: Misclassify program context for helper access
4. **Cross-Path Type Issues**: Inconsistent type tracking across execution paths

Type Confusion Attack Vectors

Attack Development

- **Type State Manipulation**: Craft programs that confuse verifier type tracking
- **Context Switching**: Exploit type inconsistencies during context changes
- **Helper Function Abuse**: Use type confusion to access restricted helpers
- **Memory Layout Attacks**: Leverage type confusion for memory corruption

JIT Compiler Exploitation

Multi-Architecture Vulnerability Research

Architecture-Specific Attack Vectors

```
class JITExploitationEngine:
    def __init__(self):
        self.architectures = {
            'x86_64': X64JITExploit(),
            'arm64': ARM64JITExploit(),
            'mips': MIPSJITExploit(),
            'riscv': RISCVJITExploit()
        }

    def find_jit_bugs(self, arch, ebpf_program):
        """
        Target architecture-specific JIT compilation vulnerabilities
        """
        return self.architectures[arch].analyze_compilation(ebpf_program)
```

Part 3

Live Exploitation Demonstration

Vulnerability Discovery and Weaponization

LIVE DEMO

State-Aware eBPF Exploitation

Vulnerability Discovery Process

What You're Witnessing

- **State-aware fuzzing**: Targeting specific verifier vulnerability classes
- **Real verifier bypass discovery**: Live bug discovery in bounds calculation logic
- **Exploitation development**: Converting verifier bug into arbitrary read/write primitive
- **KASLR bypass**: Information leak exploitation for kernel base address discovery
- **Complete privilege escalation**: From unprivileged user to root through eBPF exploitation

Defensive Recommendations

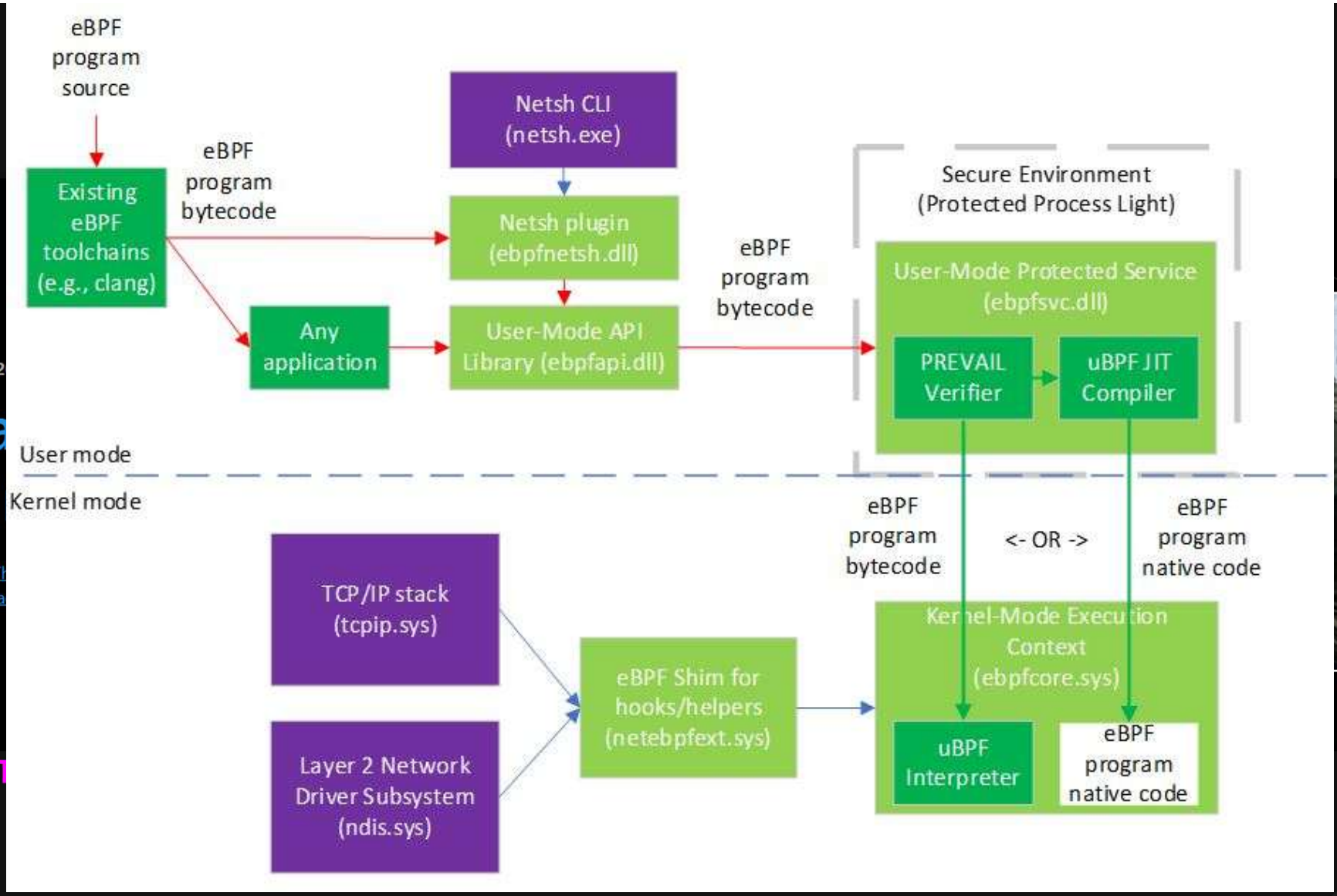
Runtime Protection

- **Helper Function Hardening**: Enhanced input validation and capability restrictions
- **Dynamic Monitoring**: Real-time detection of exploitation attempts
- **Capability Restrictions**: Fine-grained privilege separation for eBPF programs
- **Audit Infrastructure**: Comprehensive logging for forensic analysis

May 10, 2

Ma
Wi

By Dave Th
Poorna Ga



Conclusion

- **State-Aware Fuzzing Methodologies**: Improvement in vulnerability discovery efficiency
- **Systematic Exploitation Framework**: Reliable exploitation chains for modern eBPF vulnerabilities
- **Concrete Hardening Recommendations**: Actionable security improvements for eBPF subsystem

Questions

📞 Contact Information

Dr. Agostino "van1sh" Panico
📧 van1sh@securitybsides.it
🐦‍🔥 @van1sh_bsidesit
🔗 GitHub: github.com/poppopjmp/Leviathan
Research framework and documentation Open Source by the End of August 2025