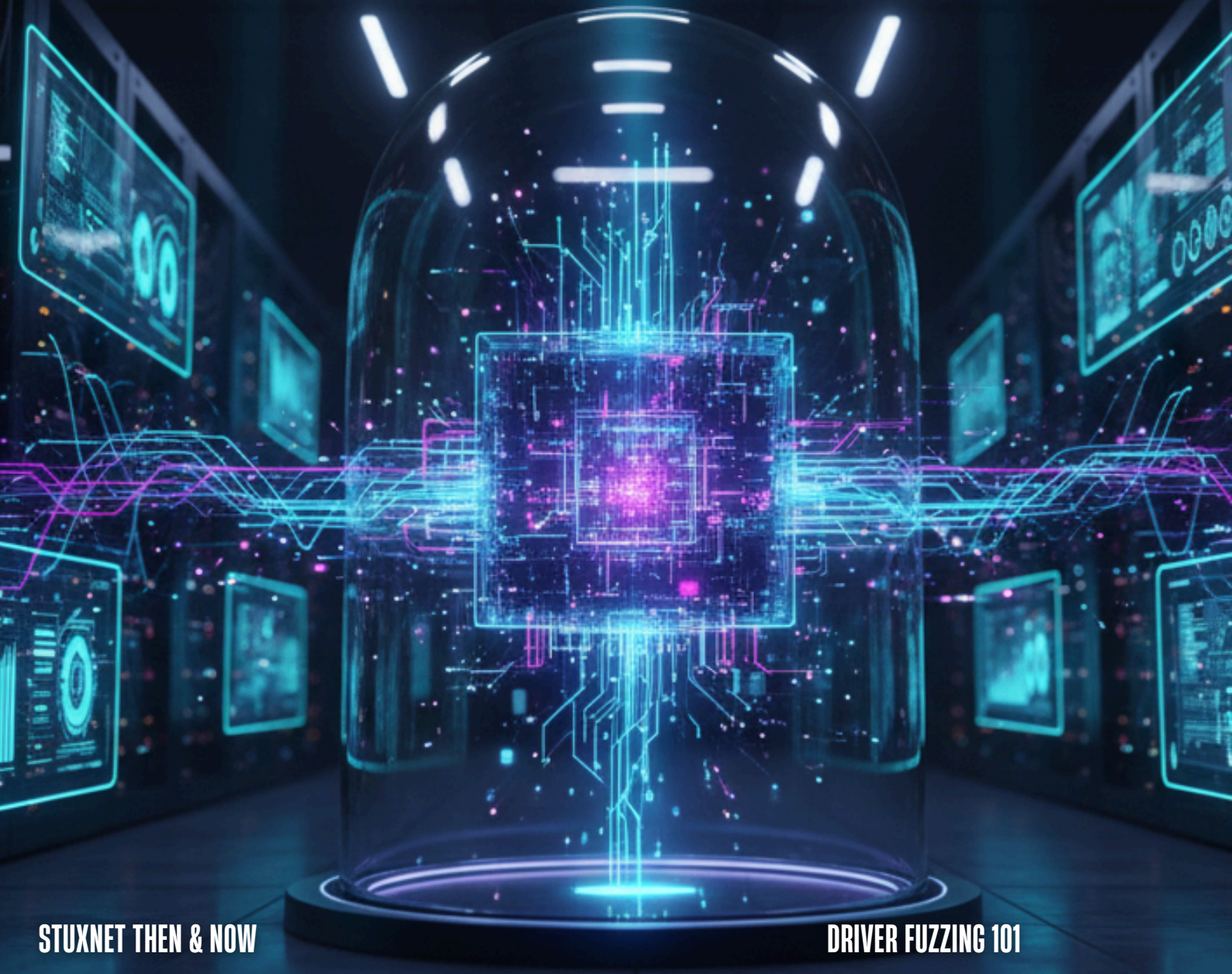
CYBER SECURITY

FRONT /> CODE

SYSTEM SECURITY EVOLUTION & ADVANCED EXPLOIT

October 2025



STUXNET THEN & NOW

NSO GROUP & PEGASUS

BRING-YOUR-OWN-VULNERABLE-DRIVER (BYOVD

DRIVER FUZZING 101

UNWINDING KERNEL STACK TRACES

“SECURE BY DESIGN„ POLICY GATHERS MOMENTUM IN THE UK

Cybersecurity 2025

October 2025/ Volume 01

1 Stuxnet Then & Now: Malware That Broke New Ground

Stuxnet (uncovered in 2010) was a watershed moment: it was the first known malware designed to cause physical destruction. This highly sophisticated worm infiltrated Iran's Natanz nuclear plant, believed to be written by nation-state actors, and directly sabotaged the industrial control systems there. Once inside, Stuxnet searched for Siemens industrial control software (used to run uranium centrifuges) and issued malicious commands. Stuxnet secretly commandeered the plant's Siemens PLC controllers and subtly tweaked centrifuge rotation speeds to induce mechanical failure (malwarebytes.com). While it was running, Stuxnet disguised its activities by replaying fake "normal" sensor readings to operators, so nobody realized the turbines were being pushed to the breaking point. In the words of cybersecurity analysts, Stuxnet was "the most aggressive cyber-physical attack ever documented" (malwarebytes.com). It proved that malware could carry a literal "warhead" using code to bend real-world physics.

The legacy of Stuxnet is everywhere in modern cybersecurity. Following its debut, similar attacks on industrial systems began to emerge. For instance, in 2016 a malware known as "CrashOverride/Industroyer" was discovered, capable of issuing shutdown commands to power grid breakers. Investigators say this tool was used to briefly black out parts of the Ukrainian electrical grid in December 2016 (reuters.com). Likewise, in 2017 the "Triton (aka Trisis)" malware hacked into safety controls of a Saudi Arabian petrochemical plant. Triton's breach of industrial safety systems was a first-of-its-kind "watershed" event: hackers could potentially have shut down the plant by deceiving safety controllers (the attackers' tools "could be fooled to indicate that everything is okay" even while the plant was being sabotaged) (reuters.com). Fortunately in that case the malware prematurely shut itself down, so disaster was averted, but the lesson was chilling.

More recently, even "regular" ransomware gangs have targeted critical infrastructure. A stark example is the Colonial Pipeline attack in May 2021. Hackers seized control of the U.S. East Coast's largest fuel pipeline, forcing it to shut down entirely for nearly a week. Colonial Pipeline paid a $4.4 million ransom to regain access, but not before the outage caused huge gasoline shortages in the Southeast (en.wikipedia.org). This incident underlined that IT-centric threats can have massive physical consequences when energy and utility networks get hit.

Despite all this, experts warn that many critical systems remain just as exposed as they were 15 years ago. At a 2025 U.S. House hearing, veteran ICS security analyst Joe Weiss bluntly observed that "critical infrastructures continue to be susceptible to Stuxnet-type attacks" (controlglobal.com). In other words, the vulnerabilities that Stuxnet exploited, trusting field sensors, unsegmented OT (Operational Technology) networks, obscure protocols, have not been fully fixed. Many industrial control systems still lack modern protections or even awareness of these threats. As Weiss noted, sophisticated hacks often "look like equipment malfunctions", so incidents can slip by undetected if operators assume it's just a sensor glitch (controlglobal.com). This remains a dangerous blind spot: an attack on a turbine might be mistaken for a hardware failure unless process-level monitoring is in place.

The good news is that awareness is finally translating into defense. Industry guidelines (like NIST's ICS security framework) now emphasize isolating OT networks from the Internet, implementing strict access controls, and closely monitoring physical processes, not just network traffic. Operators are urged to keep detailed inventories of sensors and controllers, so anomalies can not hide in the weeds. Lessons from Stuxnet and its successors have led to new tools that watch the "physical layer" of systems: for example, alarms if a centrifuge spins beyond safe limits. Public-private threat-sharing forums (e.g. ICS-CERT) exist so that operators learn quickly about new ICS malware variants. In short, defenders are moving toward a holistic view that spans software and hardware.

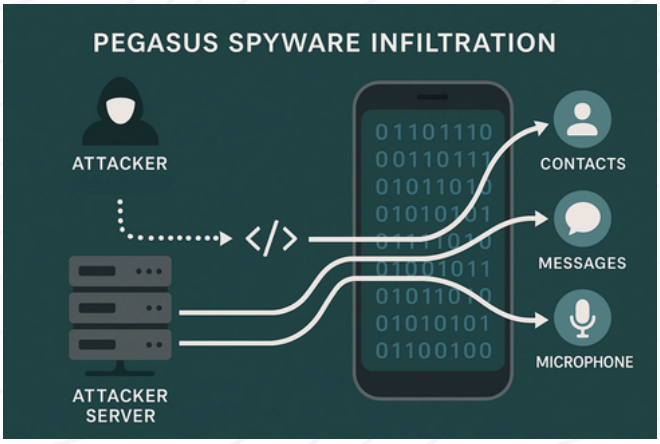
Industrial control room

In summary, Stuxnet broke unprecedented ground by showing cyber weapons can cause real-world damage. Its story reshaped cybersecurity strategy: no longer is blocking Internet intrusions enough. We must also protect the tiny devices and control loops that actually run our infrastructure. Fifteen years later, Stuxnet's impact is still unfolding, a reminder that defending against cyber-physical attacks is an ongoing mission (controlglobal.com; reuters.com).

Key Takeaways: Stuxnet was the first malware "cyber-weapon" that physically damaged equipment (malwarebytes.com). In the years since, new ICS-focused malware (Industroyer, Triton, etc.) have struck utilities and plants (reuters.com). Experts now emphasize that many industrial systems are still vulnerable, lacking simple protections and wrongly treated like ordinary IT networks (controlglobal.com). Defenses must span networks and physical processes (segmentation, sensor checks, ICS-aware monitoring). In short, Stuxnet taught us that code can have a physical "warhead," and protecting critical infrastructure means learning to think like a defender of both software and hardware (malwarebytes.com).

2 NSO Group & Pegasus: Unraveling the Spyware Scandal

Pegasus is NSO Group's notorious spyware that can stealthily hijack smartphones using zero-click exploits; no user action needed; and harvest virtually everything on the device: texts, calls, location, camera, microphone, you name it (amnesty.org). NSO Group is an Israeli cybersecurity firm that develops surveillance tools, primarily marketed to governments for law enforcement and national security purposes. Originally sold to fight terrorism and crime, forensic reports show it was widely abused by governments. Investigations revealed that state clients around the globe, from Saudi Arabia and Mexico to Poland and El Salvador were using Pegasus to spy on journalists, activists and dissidents (reuters.com; reuters.com). In fact, the Pegasus Project (a 2021 media investigation,involved collaboration among 17 media organizations, led by Forbidden Stories, to analyze a leaked list of potential surveillance targets, exposing the scale of Pegasus misuse) exposed a leaked list of over 50,000 phone numbers including world leaders and reporters, across 50+ countries as potential surveillance targets (amnesty.org). That scandal prompted global outrage. And Pegasus is not just historical news: Amnesty International's tech lab recently confirmed that two prominent Indian journalists were hacked with Pegasus in late 2023 (amnesty.org), showing this invasive tool is very much alive and being used today.



A conceptual diagram of a Pegasus spyware attack

Detecting and preventing Pegasus spyware is challenging due to its advanced nature and the use of zero-click exploits. Traditional antivirus software may not be effective against such sophisticated threats. However, specific tools like the Mobile Verification Toolkit (MVT) developed by Amnesty International can analyze mobile devices for indicators of compromise related to Pegasus. Additionally, Apple has introduced Lockdown mode in iOS 16 to reduce the attack surface, making it harder for spyware to exploit vulnerabilities. Users should keep their devices updated, use secure communication channels, and be cautious with app permissions to minimize risks (us.norton.com).

Legal Backlash and Accountability: After years of secretive abuse, NSO is finally under fire. In May 2025, Meta (WhatsApp's parent) won a $168 million jury verdict against NSO (reuters.com). U.S. courts found NSO had secretly exploited a WhatsApp bug to install Pegasus on users' phones. The verdict awarded about $444K in compensatory damages plus $167M in punitive fines (reuters.com). Reuters reports that NSO is now "a poster child for the surveillance industry and their abuses and impunity," long arguing its tools target only terrorists and pedophiles while evidence showed its software was tied to widespread spying (reuters.com). Trial testimony even revealed NSO had a 140-person R&D team with a $50M budget for hacking phones and recorded government customers such as Uzbekistan, Saudi Arabia and Mexico (reuters.com). Apple has sued NSO too (in late 2021) for similar allegations that U.S. iPhones were breached by Pegasus (reuters.com). Apple's lawsuit seeks to hold NSO accountable for targeting iPhone users and aims to set a precedent for restricting spyware misuse. These landmark cases signal that cyber-spyware vendors can be held legally accountable for abuses.

Pegasus spyware

Policy and Regulation: The Pegasus saga has spurred swift policy action. The U.S. Commerce Dept. formally blacklisted NSO in 2021, banning U.S. exports to NSO as punishment for its "malicious" spyware sales to foreign governments (commerce.gov). In Europe, lawmakers have opened inquiries into Pegasus use, the EU Parliament even set up a special committee to investigate reports that Pegasus and similar spyware were used against EU citizens and leaders (politico.eu). Meanwhile NGOs are demanding stricter controls. Human Rights Watch warns that governments "should urgently suspend sales and transfers" of such spyware until proper human-rights-protecting oversight is in place (hrw.org). Amnesty International and other groups have similarly called for export bans or licenses revocation, emphasizing that unchecked surveillance tools violate human rights. The bottom line: many experts now say our laws and norms have not kept up with these intrusions. As one researcher put it, Pegasus reminds us that code can have physical "warheads," so without new ethical rules and regulations our democracies and privacy are at risk (hrw.org; commerce.gov).

The commercialization of advanced surveillance tools like Pegasus has created a lucrative market, with governments paying between $3 million and $30 million for access to such capabilities, as revealed in trial testimony. This high price reflects the tool's sophistication and comprehensive surveillance features. However, the financial incentives also encourage the proliferation of these technologies, potentially leading to increased misuse and human rights violations. Consequently, there is an urgent need for stricter regulations and oversight to ensure that surveillance tools are used ethically and in accordance with legal standards (lookout.com).

In summary, Pegasus taught a hard lesson: unrestrained digital surveillance erodes trust. The NSO/WhatsApp trial and international scrutiny show the tide is turning toward accountability. Moving forward, To prevent future abuses both governments and private tech companies will need clear, enforceable rules for any hacking tools, or face losing public trust and legal battles.

3 Bring-Your-Own-Vulnerable-Driver (BYOVD): Hijacking Legitimate Binaries

Bring-Your-Own-Vulnerable-Driver (BYOVD) is a technique where attackers drop and load a trusted but vulnerable, signed kernel driver into a Windows system. Since the driver is legitimate and signed, the operating system accepts it. The attacker then exploits the driver's flaw to perform actions at kernel level.
Kernel drivers run with ring-0 privileges. That lets attackers read or write memory, escalate to SYSTEM, disable protections, or load unsigned code. This way, they hijack trusted binaries to bypass security.
BYOVD is gaining popularity among threat actors because it offers high stealth and low detection rates. It is different from traditional malware because these drivers slip past most defenses due to their signed status.

Why are BYOVD Attacks Effective?

One reason BYOVD works so well is that many organizations rely on signature-based detection. These drivers are already signed and trusted, making them less likely to raise red flags.

Trusted binaries are whitelisted

Signed drivers pass Windows Driver Signature Enforcement. These binaries are often trusted by EDR and antivirus solutions. As a result, using a vulnerable driver is more covert than deploying malware.

Kernel-level access
Once loaded, the driver runs in kernel mode. It can escalate privileges, disable security features, patch MSRs, or overwrite raw disk sectors.

How do attackers exploit vulnerable drivers?

BYOVD attacks follow a clear pattern:

1. Drop a signed, vulnerable driver – often reused from legitimate software.
2. Load the driver – using admin rights or service creation.
3. Exploit memory or control flaws – for privilege escalation or disabling security services.
4. Hijack trusted binaries – such as EDR components or Windows processes.
5. Persist or drift laterally – by planting rootkits, dropping ransomware, or mining cryptominer.

BYOVD Attacks: Real World Examples

Many high-profile campaigns have leveraged BYOVD tactics. These include ransomware groups and nation-state actors who exploit the technique for stealth and persistence.

RobbinHood (Gigabyte driver)
In 2019, RobbinHood ransomware abused the gdrv.sys driver from Gigabyte. It exploited CVE-2018-19320 to disable Driver Signature Enforcement, then installed its own malicious driver.

BlackByte (MSI Afterburner)
BlackByte used RTCore64.sys, the MSI Afterburner driver (CVE-2019-16098), to terminate over 1,000 security-related processes.

Lazarus (Dell DBUtil)
North Korea's Lazarus APT abused dbutil_2_3.sys (Dell's firmware driver, CVE-2021-21551) in 2021. The driver loaded a rootkit and disabled ETW, filesystem, registry and process tracing.

RansomHub (EDRKillShifter tool)
In 2024, RansomHub launched EDRKillShifter. It loaded a vulnerable driver via a password-protected loader. It then exploited and shut down EDR agents before executing ransomware.

GHOSTENGINE (Cryptominers)
The REF4578 campaign used aswArPot.sys (Avast) and IObit's unlocker driver to kill EDR processes and install a cryptominer called XMRig.

How does this Hijack Legitimate Binaries?

Hijacking occurs when vulnerable drivers are used to alter or terminate legitimate security binaries. Attackers use kernel privileges to tamper with system integrity.

Terminate or alter security processes

Vulnerable drivers may allow arbitrary kernel memory writes. Attackers use these to pause or kill security process threads. This disables protections silently.

Load unsigned code into kernel

Some drivers allow mapping of unsigned code. Attackers exploit this to load rootkits, unsigned drivers, or alter MSRs.

Bypass Code Integrity

Drivers that manipulate MSR or IOPORT registers can disable Code Integrity and HVCI. This pits Windows into loading unauthorized drivers freely.

Detection and Mitigation Strategies

Blocking BYOVD requires both proactive and reactive steps. Endpoint monitoring, driver control policies, and behavior analytics all play a role.

1. Driver whitelisting and auditing

Keep an up-to-date whitelist of approved drivers and review it regularly. Use configuration management tools and endpoint monitoring to detect drivers that aren't authorized or are no longer in use.
This allows only signed and verified drivers run on the system. Learn more about maintaining endpoint security through driver validation practices.

2. Patch or block vulnerable drivers

It is needed to monitor trusted sources like CVE databases and hardware vendor bulletins for driver-related security advisories.
Once a vulnerability is disclosed, either patch it immediately or block the driver using Group Policy or endpoint protection tools. Quick response limits the time attackers can exploit the flaw. See how to manage known driver vulnerabilities.

3. Harden driver loading

Another strategy is to restrict who can load new drivers by minimizing admin-level access. You can use features like Windows Defender Application Control (WDAC) and enforce kernel-mode code integrity policies. These tools help block unsigned or unapproved drivers from loading during runtime.

4. Monitor kernel telemetry

You can use tools like Windows Event Viewer, Sysmon or other commercial telemetry platforms to log kernel-level events.
Another thing is to set alerts for unexpected driver loads such as unsigned or outdated ones and watch for suspicious IOCTL behavior that could signal exploitation attempts. Also learn how telemetry insights support real-time detection.

5. Use behavior-based detection

Signature-based solutions alone won't catch BYOVD attacks. Use EDR or XDR platforms with behavioral analytics to identify anomalies like unusual driver installs or EDR agent tampering. Monitoring patterns linked to driver execution helps flag suspicious activity.

6. Contain suspected compromises quickly

If you suspect a BYOVD incident, isolate the device from the network immediately to stop lateral movement. Perform memory analysis if needed.
If the system's kernel integrity is compromised, then a full re-image is often the safest route. This helps to make sure that the threat is fully removed.

To Conclude

BYOVD attacks hijack legitimate, signed drivers to achieve kernel-level control. They bypass WDSE and many security tools. Attackers use them to disable protections, escalate privileges, or load rootkits.
Organizations face significant risk from BYOVD. Frequent vulnerabilities exist in gaming, printer, storage, and firmware driver packages.
To defend against BYOVD, teams should enforce driver whitelisting, timely patching, strict privilege management, telemetry, and behavior monitoring. These steps help detect and block driver-based hijacks before they compromise binaries at kernel level.

4 Driver Fuzzing 101: Modern Tools to Catch Bugs Before Hackers Do

Driver fuzzing is a growing technique used to identify hidden vulnerabilities in some of the most critical parts of a system: kernel drivers.
These drivers manage communication between the operating system and hardware. Because they run with high privileges, even minor bugs can lead to major consequences like system crashes or privilege escalation.
Instead of waiting for attackers to discover these flaws, security teams now rely on driver fuzzing to simulate unusual or invalid inputs. This helps reveal edge-case bugs that would likely go unnoticed in traditional testing. Fuzzing is fast, repeatable, and highly effective at catching problems early.

What is Driver Fuzzing?

Fuzzing is an automated testing technique that feeds invalid or random data to a program and observes its behavior for crashes or memory violations.

Driver fuzzing targets operating system drivers, critical code that runs in kernel space. Bugs here can lead to major security breaches.

Why Fuzz Drivers?

Drivers handle inputs from devices and user processes. A malformed ioctl request or unexpected USB input can bypass error checks in complex validation logic. Fuzzing exposes these corner cases before they become exploited.

Categories of Driver Fuzzing

Driver fuzzing falls into multiple categories depending on the approach. Mutation-based fuzzers take existing valid inputs and slightly alter them to trigger unusual code paths.

Generation-based fuzzers create structured inputs from scratch using known protocols or file formats.

Mutation-based vs. Generation-based

- Mutation-based fuzzers modify valid inputs for minor variations.
- Generation-based fuzzers build structured inputs from scratch.

Both approaches are used in driver fuzzers, depending on whether input formats (like USB packets or ioctl structures) are known.

White-grey-black-box Techniques

- **White-box** fuzzers use source code and symbolic execution.
- **Grey-box** fuzzers rely on coverage feedback (e.g., KCOV).
- **Black-box** fuzzers run without code visibility.

Kernel and driver fuzzing usually uses grey-box or white-box approaches to guide input generation and improve efficiency.

Leading Driver Fuzzing Tools

Each tool listed below has specific use cases, from Linux kernel modules to Windows IOCTL fuzzing. Some tools support symbolic execution, while others rely on emulation or runtime feedback.

1. Syzkaller

Syzkaller is a robust kernel fuzzer for Linux. It uses coverage feedback (via KCOV) and integrates sanitizers to catch memory issues. Security teams rely on it for high-volume fuzzing of Linux syscalls and drivers.

2. OneFuzz

Microsoft's OneFuzz is an ensemble fuzzing as a service platform supporting Windows and Linux. It automates build, crash triage, and debugging. While not driver-specific, Windows kernel code is fuzzed using this tool to uncover driver flaws.

3. IOCTLbf

IOCTLbf is a proof-of-concept tool for Windows drivers. It scans ioctl codes and performs generation-based fuzzing, even when no real applications use those ioctls.

4. difuze

Developed by the UCSB SecLab, difuze uses LLVM-based static analysis to recover driver interfaces and manage fuzzed inputs. It automates both interface discovery and fuzzing, targeting Linux kernel modules.

4 Driver Fuzzing 101: Modern Tools to Catch Bugs Before Hackers Do

5. DEVFUZZ
DEVFUZZ builds device models for drivers using symbolic execution. It then guides fuzzers without requiring physical hardware. Tested on Linux, FreeBSD, and Windows, it uncovered millions of code paths and dozens of vulnerabilities.

6. USBFuzz
USBFuzz emulates USB devices and feeds data to host drivers under test. Its modular design supports multiple platforms, making it ideal for fuzzing driver interfaces that depend on USB devices.

7. Agamotto
Agamotto boosts kernel fuzzing performance using VM checkpointing. It speeds up state restoration scenarios, enhancing fuzzing throughput for USB and PCI drivers.

8. SyzParam
SyzParam is a 2025 breakthrough. It dynamically extracts and mutates kernel module parameters, improving code coverage by over 30% and revealing dozens of new bugs.

Supporting Tools in Fuzzing

Sanitizers

Sanitizers detect memory errors during fuzzing. KernelAddressSanitizer (KASan) works in Linux and Windows to catch out-of-bounds and use-after-free bugs.

AFL and AFL++

AFL is a genetic algorithm-based fuzzer. It's foundational, and its forks (like AFL++) are used with syscall or driver fuzzers to optimize mutation performance.

Best Practices in Driver Fuzzing

Effective driver fuzzing requires more than just pointing tools at a target. Well-structured practices help reveal more bugs, reduce noise, and improve reproducibility.

1. Automate Interface Discovery
Use tools like difuze or IOCTLbf to scan driver code or binaries and extract supported ioctl codes and structures. Auto-discovery ensures you cover more input paths than manual enumeration. When interfaces change, repeat discovery to stay current.

2. Use Sanitizers and Coverage Feedback
Compile drivers with KASan, UBSan, and KCOV. These sanitizers reveal memory and undefined behavior errors, while coverage guidance steers fuzzers to unexplored code paths. For example, Syzkaller's integration with KCOV is key to its effectiveness.

3. Leverage Emulation, VM Checkpointing, and Device-free Fuzzing
Use QEMU or TUN/TAP for driver execution environments. Tools like USBFuzz and Agamotto speed fuzz cycles by restoring VM snapshot states instead of rebooting.

Semantic-driven tools like DR.FUZZ enable fuzzing without actual hardware, dramatically expanding target coverage.

4. Incorporate Runtime Parameters and Structured Initialization
Introducing runtime parameters via SyzParam improves reach into complex code paths. Tools should extract module parameters and mutate them intelligently.
Additionally, maintain valid driver initialization using pseudo-syscalls and literal C code to set up required states before fuzzing begins

5. Write High-Quality Fuzz Drivers
Drivers that wrap calls, set up state, and clean resources lead to better fuzzing results. Applying methodologies like PromptFuzz for semi-automated generation helps ensure structure and error-free fuzz driver code.

4 Driver Fuzzing 101: Modern Tools to Catch Bugs Before Hackers Do

6. Run Multiple, Time-Bound Trials
Fuzzing can vary due to randomness. Research shows running several trials (e.g., 3x24 hours) yields more reliable bug discovery and performance metrics. Allocate fixed windows (overnight, weekends) to balance test coverage with development flow.

7. Use Error-State and Fault-Injection Feedback
Incorporate coverage-guided fault injection to explore error handling paths. Tools like FIZZER identify static failure sites and fuzz error sequences to hit deeper code conditions.

8. Automate Crash Triage and Management
Use crash deduplication and automated triage tools (e.g., OneFuzz, OSS-Fuzz pipelines) to assess severity, reproduce reliably, and integrate fixes into defect trackers or CI systems

Typical Fuzzing Workflow

Here is a structured workflow to catch driver bugs early:

1. Setup test environment (VM or test machine).
2. Discover Interfaces with tools like difuze or IOCTLbf.
3. Generate Inputs using Syzkaller, DEVFUZZ, USBFuzz.
4. Instrument Sanitizers during build.
5. Fuzz at Scale, utilizing Agamotto checkpointing for efficiency.
6. Triage crashes via OneFuzz or custom scripts.
7. Report and Patch validation issues before merging driver updates.

4 Driver Fuzzing 101: Modern Tools to Catch Bugs Before Hackers Do

Common Risks and How Fuzz Tools Mitigate Them

Driver bugs can lead to memory corruption, denial-of-service, or even privilege escalation. Fuzzing tools mitigate these risks by continuously stress-testing the code under edge conditions.

Risk Type	Example Threat	Mitigation Tool/Method
Memory corruption	Buffer overflow in ioctl	KASan + Syzkaller
Unsupported IOCTL codes	Unhandled ioctl leads to crash	IOCTLbf
Hardware edge cases	Rare USB packet structure	USBFuzz emulation
Deep driver paths	Complex init logic using MMIO/PCI	DEVFUZZ symbolic execution + fuzzing
Stateful behavior	Module parameters set at runtime	SyzParam parameter coverage
Slow fuzz cycles	Long init times for each case	Agamotto checkpoint technique

To Conclude

Driver fuzzing is a much-needed defense layer. Tools like Syzkaller, IOCTLbf, difuze, DEVFUZZ, USBFuzz, Agamotto, and SyzParam allow teams to find kernel-level bugs early.
Supporting frameworks such as OneFuzz and sanitizers improve crash finding and triage. Implementing a systematic fuzzing pipeline helps prevent vulnerabilities from entering production systems.

5 Unwinding Kernel Stack Traces in Obfuscated Driver Environments

Classical stack unwinding always breaks when kernel drivers are heavily obfuscated or use new packing and hiding mechanisms, as they can. It is not just the lack of symbols: interrupted unwind metadata, corrupted frame contexts, non-standard and non-standard trap handlers, and control flow.

This article takes a closer look at the mechanics and reliable techniques of recuperating call chains even when regular unwind formats do not work.

Why Obfuscation Breaks Stack Unwinding?

Kernel-mode driver obfuscation applications are uncommon, though existent, particularly in special IP-protection or rootkit applications. Standard stack tracing using frame pointers, ORC tables, or DWARF cannot work when code is packed, that is, generated dynamically, or does not have typical prologue/epilogue conventions.

The documentation of Microsoft Kernel-mode Hardware-enforced Stack Protection notes the possibility that drivers written with control-flow obfuscation are incompatible with shadow-stack enforcement, making it clear that these kinds of transformations make traceability impossible.

Frame Pointers, ORC, and When They Stall

Most programs that run on Linux use frame pointers or ORC unwind tables to unwind. However, the mechanism is less reliable when obfuscated or compiled away frame pointers.

This can be observed in the very documentation of the kernel itself, which warns that an architecture cannot accept unreliable code paths, i.e., where memory operations are out of control, e.g., trampolines, or foreign code such as eBPF: what obfuscation does: obscures or eliminates stack-safe areas.

On the same note, ORC provides assistance when the module respects compiler-generated unwind metadata. However, a corrupt packer can discard or corrupt those tables completely.

Hypervisor-based Introspection: Reversing Memory Assumptions

A single direction seems to be one of the brightest questionings: The Reversing Machine (TRM), a hypervisor-based memory introspection system. TRM can rebuild complete memory maps of the kernel and modules during execution, track both kernel/user transitions, and rebuild structures and execution flows within obfuscated environments.

According to its designers, it reduces by 75 % the manual reverse-engineering time of obfuscated kernel modules.

Unlike stack unwinding, this method completely avoids unwinding because it reconstructs the execution context based on the raw memory patterns and control-flow transitions.

Dynamic Profiling of Kernel Events

It basically provides the possibility to use kernel telemetry and ETW (Event Tracing for Windows) to take runtime call-stack samples, e.g., even in case of their obfuscation, without symbols or frame layouts. In their Elastic 8.11 release, Elastic Security Labs added more support to ETW in the kernel, where they record in-memory call stacks to find evasive behavior.

Why is this useful? Although one may obfuscate the functions, the context of the driver callback and dispatch routines is usually the same across programs, and it is possible to cluster and trend patterns within them.

Detecting Invalid Unwinds and Stack Corruption

When an illegal boundary of the stack pointer is reached other than via corrupted frames or bad stack pointers (a frequent condition under a corrupted driver or kernel in general, or obfuscation in particular), the driver may crash with bug check UNWIND_ON_INVALID_STACK (0x1AB).

Crash dumping with this bug check flag will indicate where unwind failed, and show the limits of valid and invalid stack states.

Techniques to Reassemble Traces in Practice

1. Pre-warm trace points in safe code paths

Before the obfuscated part, lightweight instrumentation is inserted at known points of kernel entry or kernel exit. These serve as anchor frames- even if you lose some deeper in, you can use the anchors and assemble call slices outward.

2. Use hypervisor memory snapshotting

The hypervisor pauses or suspends the VM, captures physical memory pages, particularly those of the driver module, and exposes them through LibVMI's shared memory interface to the introspection application.

That application then analyzes instruction boundaries, cross-mode transitions, and call addresses. By applying memory-access and control-flow heuristics in conjunction with driver base address info, it reconstructs an approximate call sequence from obfuscated modules, even without conventional unwind metadata.

3. Heuristic-based return address hunting

Return addresses will generally push onto the stack in a predictable format, even without the frame pointers. You may also scan stack memory to seek likely return addresses within the driver's virtual address window and utilize these as unwind hints. Merge with control-flow graphs that are statically extracted from unpacked code.

4. ETW / ETW-like callback capture

In Windows, call register kernel notification callbacks (e.g., PsSetCreateProcessNotifyRoutineEx, ObRegisterCallbacks), which EDR/AV drivers usually use to record entry/exit of some important routines. They all represent practical landmarks that assist in confining and checking the restored stack frames.

Future Directions

Future research highlights challenges like continuous KASLR and runtime re-randomization, as demonstrated by projects like Adelie, which make static mapping and return-address heuristics increasingly unreliable.

Bottom Line

Unwinding kernel stack traces in obfuscated drivers demands more than static debugging. Traditional methods fall short with KASLR, shadow stacks, and driver isolation.

Techniques like memory introspection, ETW tracing, and return address heuristics are now essential. It is becoming the rule and not the exception. Adaptive, context-aware unwinding is not a choice; it is a necessity for researchers and security engineers to be able to keep up with the constantly changing protections and ever more obscured execution paths in the kernel.

6 “Secure by Design” Policy Gathers Momentum in the UK

The UK government is now requiring all central departments and arm’s length bodies to adopt the Secure by Design policy. Introduced in May 2025, this move sets a new baseline for how digital systems must be built across the public sector.

The policy applies to system delivery teams, digital leaders, developers, and commercial officers. It demands early security planning, continuous risk assessment, and full compliance with the National Cyber Security Centre’s Cyber Assessment Framework (CAF).

Defence Took the First Step

The Ministry of Defence had already adopted Secure by Design before the rest of government followed. It published a "problem book" describing technical debt, legacy constraints, and supplier challenges. The Defence Digital team then formalised seven Secure by Design principles.

These include clearly defining the system context, managing risk across the lifecycle, and creating audit trails that can be tested by third parties. The ministry expects defence suppliers to build these into their delivery models.

Code of Practice for Software Vendors

In March 2025, the UK government released a Software Security Code of Practice. It lays out 14 principles that software vendors must follow when providing systems to the public sector.

The principles cover:

- Secure development environments
- Testing for known vulnerabilities
- Managing third-party dependencies
- Incident response planning

The code aims to bring vendor standards in line with Secure by Design expectations.

Parliament and Watchdogs Back the Push

The Public Accounts Committee has called for stronger cyber resilience in the public sector. In its latest report, it said too many departments rely on outdated defences and patch fixes. The committee backed Secure by Design as a long-term strategy to improve system safety.

The National Audit Office has raised similar concerns. It found that legacy IT and fragmented oversight make systems more vulnerable. Both groups see the new policy as a necessary shift.

New Laws Add Pressure

Secure by Design fits into a broader set of legal reforms. The Cyber Security and Resilience Bill, introduced in 2024, is now moving through Parliament. It will bring new reporting rules and stronger oversight of cyber incidents.

The Online Safety Act, which takes full effect on 25 July 2025, targets digital platforms. It forces companies to build user protections into their systems at the design stage. Failure to comply can lead to enforcement action.

Implementation Support and Tools

The Central Digital and Data Office (CDDO) has published step-by-step support for departments. It encourages teams to form delivery groups with both technical and policy staff. A readiness checklist and transition planning guide are available.
Weekly support calls are open to teams across government. These sessions help departments apply Secure by Design in procurement, cloud deployments, and agile software projects.

Barriers Slow Down Rollout

Some departments are struggling to keep pace. Reports show that security is still treated as an afterthought in many public services. Many teams lack the technical capacity to design secure systems from the ground up.

The Ministry of Defence has raised other problems. Export controls make it hard to share technical designs with suppliers. Some systems also depend on old software that cannot easily be rebuilt.

6 “Secure by Design” Policy Gathers Momentum in the UK

Software Suppliers Face Pressure

Private vendors now face pressure to meet Secure by Design and code of practice standards. Larger firms are updating internal workflows and hiring security-focused engineers. Smaller vendors face resource gaps.

Some suppliers say they need more clarity from government buyers. Others warn that the cost of meeting new standards could put pressure on pricing and delivery timelines.

Security Now Starts with Design

Secure by Design has become a formal part of how the UK government expects systems to be built. It no longer treats security as a last step. Instead, it requires developers and policy teams to plan for it at the start of every project.

The approach now influences procurement, regulation, and delivery. Whether in digital services or defence systems, teams are expected to follow the same rules. Departments that do not comply may face audits or intervention. Vendors that fall short may be excluded from future contracts.

The rules are now active. The deadlines are near. The priority is clear.