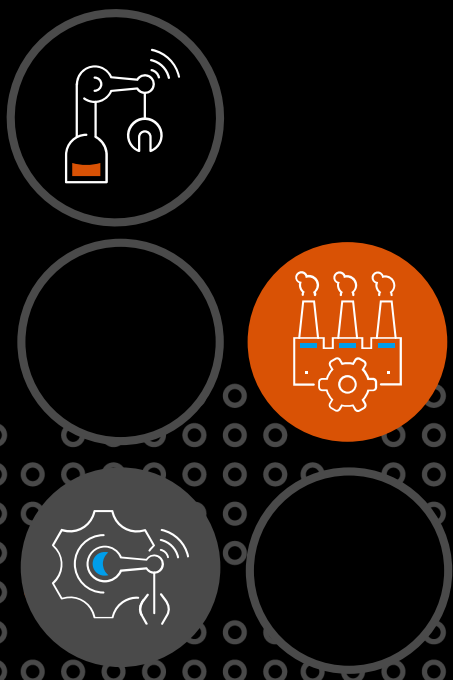
FORESCOUT

FORESCOUT | JFrog

# INFRA:HALT

## Jointly discovering and mitigating large-scale OT vulnerabilities

By Forescout Research Labs & JFrog Security Research



**Forescout Research Labs**
Daniel dos Santos
Stanislav Dashevskyi
Amine Amri
Jos Wetzels

**JFrog Security Research**
Asaf Karas
Shachar Menashe
Denys Vozniuk

# Contents

# 1. Executive summary

- In the fourth study of Project Memoria – INFRA:HALT – Forescout Research Labs and JFrog Security Research jointly disclose a set of 14 new vulnerabilities affecting the NicheStack TCP/IP stack (also known as InterNiche stack).
- NicheStack is used by several devices in the Operational Technology (OT) and critical infrastructure space, such as the popular Siemens S7 line of PLCs. Other major OT device vendors, such as Emerson, Honeywell, Mitsubishi Electric, Rockwell Automation, and Schneider Electric, were mentioned as customers of InterNiche, the original developers of the stack. Due to this popularity in OT, the most affected industry vertical is Manufacturing.
- The new vulnerabilities **allow for Remote Code Execution, Denial of Service, Information Leak, TCP Spoofing, or DNS Cache Poisoning.**
- Forescout Research Labs and JFrog Security Research exploited two of the Remote Code Execution vulnerabilities in their lab and show the potential effects of a successful attack.
- General recommended mitigations for INFRA:HALT include **limiting the network exposure of critical vulnerable devices** via network segmentation and patching devices whenever vendors release patches. Some of the vulnerabilities can also be mitigated by blocking or disabling support for unused protocols, such as HTTP.
- Many of the vulnerabilities were found **by using state-of-the-art automated binary analysis**, which paves the way for future large-scale vulnerability finding and mitigation.
- INFRA:HALT confirms earlier findings of Project Memoria, namely similar vulnerabilities appearing in different implementations, both open and closed source. In fact, INFRA:HALT includes examples of memory corruption like in AMNESIA:33, weak ISN generation like in NUMBER:JACK and DNS vulnerabilities like in NAME:WRECK.
- INFRA:HALT extends the community understanding of vulnerability patterns and issues related to IoT/OT software supply chains. In this report, we discuss lessons learned and provide suggestions on what the community can do to mitigate these emerging threats.

INFORMATIONAL

# A Recap on TCP/IP stacks and Project Memoria

A TCP/IP stack is a piece of software that implements basic network communication for all IP-connected devices, including Internet of Things (IoT), operational technology (OT) and information technology (IT). Not only are TCP/IP stacks widespread; they also are notoriously vulnerable due to (i) codebases created decades ago and (ii) an attractive attack surface, including protocols that cross network perimeters and lots of unauthenticated functionality.

Noticing the impact of these foundational components, Forescout Research Labs has launched Project Memoria with the goal of collaborating with industry peers and research institutes to provide the cybersecurity community with the largest study on the security of TCP/IP stacks.

The latest examples of TCP/IP stack vulnerabilities include:

- Ripple20, a set of 19 vulnerabilities on the Treck TCP/IP stack disclosed by JSOF in June 2020. Forescout Research Labs worked in close collaboration with JSOF to identify vendors and devices potentially affected by Ripple20.
- AMNESIA:33, a set of 33 vulnerabilities affecting four open-source TCP/IP stacks disclosed in December 2020 by Forescout Research Labs.
- NUMBER:JACK, a set of nine vulnerabilities affecting the Initial Sequence Number (ISN) implementation in nine TCP/IP stacks disclosed in February 2021 by Forescout Research Labs.
- NAME:WRECK, a set of nine vulnerabilities affecting DNS clients of four TCP/IP stacks disclosed in April 2021 by Forescout Research Labs and JSOF.
- INFRA:HALT, a set of 14 vulnerabilities affecting InterNiche's NicheStack, disclosed in August 2021 by Forescout Research Labs and JFrog Security Research.

# 2. Main Findings

## 2.1. What is NicheStack?

NicheStack (also known as InterNiche stack) is a **proprietary TCP/IP** stack developed originally by InterNiche Technologies and acquired by HCC Embedded in 2016. The earliest copyright messages indicate that the stack was created in 1996, although InterNiche was founded in 1989. The stack was extended to support IPv6 in 2003.

In these more than two decades, the stack was distributed in several "flavors" by OEMs such as STMicroelectronics, Freescale (NXP), Altera (Intel) and Microchip for use with several (real-time) operating systems (RTOS) or its own simple RTOS called NicheTask. It also served as the basis for other TCP/IP stacks, such as SEGGER's emNet (formerly embOS/IP).

Figure 1 shows an overview of the components of the stack, including the IPv4 and IPv6 versions. These components were packaged as different product offerings by InterNiche: IPv4, IPv6, IPv4/v6 and Lite, as shown in Figure 2.

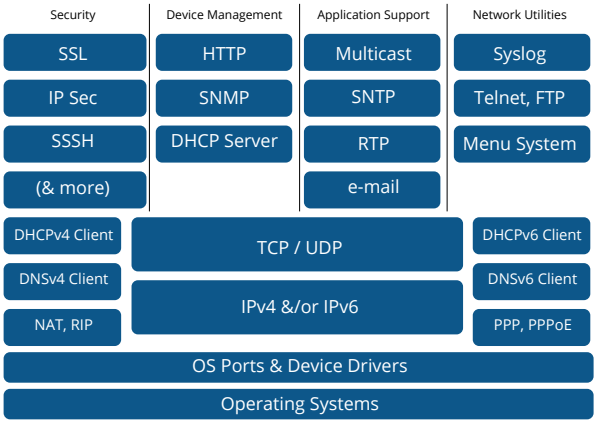



*Figure 1 – NicheStack components [readapted]*

**NicheStack Products**

| IPv6 | IPv4 | v4/v6 | Lite | |
|---|---|---|---|---|
| | ✓ | ✓ | | Auto-IP |
| ✓ | ✓ | ✓ | ✓ | DDNS |
| | ✓ | ✓ | ✓ | DHCP Server |
| ✓ | ✓ | ✓ | ✓ | DNS Client |
| | ✓ | ✓ | | DNS Server |
| ✓ | ✓ | ✓ | ✓ | Email Alerter |
| ✓ | ✓ | ✓ | ✓ | FTP Server + Client |
| | ✓ | ✓ | | IP Multicast |
| ✓ | ✓ | ✓ | | IPSec/IKE |
| | ✓ | ✓ | | NAT |
| ✓ | ✓ | ✓ | | NAT-PT |
| ✓ | ✓ | ✓ | ✓ | POP3 |
| ✓ | ✓ | ✓ | ✓ | PPP |
| ✓ | ✓ | ✓ | ✓ | ▪ MS-CHAP |
| ✓ | ✓ | ✓ | | ▪ MultiLink |
| ✓ | ✓ | ✓ | | ▪ PPPoE |
| | ✓ | ✓ | | RIP |
| | ✓ | ✓ | | SSL/TLS |
| ✓ | ✓ | ✓ | ✓ | SNMPv1 |
| ✓ | ✓ | ✓ | ✓ | SNMPv2(c) |
| ✓ | ✓ | ✓ | ✓ | SNMPv3 |
| ✓ | ✓ | ✓ | ✓ | Telnet Server |
| ✓ | ✓ | ✓ | ✓ | TFTP Client+Server |
| ✓ | ✓ | ✓ | ✓ | Web Server |
| ✓ | ✓ | ✓ | ✓ | ▪ HTML Compiler |
| | ✓ | ✓ | | ▪ Web Server-SSL |
| ✓ | ✓ | ✓ | ✓ | NicheTool |

*Figure 2 – NicheStack product offering [from nxp.com]*

### 2.2. Why analyze NicheStack?

We chose to investigate NicheStack because of:

- Its known uses in the Operational Technology and critical infrastructure space. For instance, the stack is used in Siemens S7 PLCS, which are the most popular PLCs in the world by market share.
- The lack of previous public security research done on the stack. The only relevant vulnerabilities we found are mentioned in Table 1, but they are issues that affect several stacks, which indicates a lack of focused analysis on this stack.

This led us to hypothesize that a deeper analysis could uncover similar vulnerabilities as those found before in Project Memoria.

| CVE ID | Description | Comment |
|---|---|---|
| CVE-2004-0230 | TCP, when using a large Window Size, makes it easier for remote attackers to guess sequence numbers and cause a denial of service by repeatedly injecting a TCP RST packet. | This vulnerability affects several stacks, not only NicheStack. It is related to NUMBER:JACK since it involves TCP spoofing by guessing sequence numbers (ISN). However, instead of weak ISN generation, it originates from the use of large Window Size. |
| CVE-2019-19300 | The stack can be forced to make resource-intense calls for every incoming packet, which can lead to a denial of service. Variant of SegmentSmack. | This is an instance of the SegmentSmack vulnerability that was originally found on Linux in 2018 and later also on IPnet/Vxworks. |

*Table 1 – Previously known vulnerabilities on NicheStack*

### 2.3. Analysis and findings

We had access to two versions of NicheStack for our analysis: source code of v3 (publicly available via a website exposing the source files for an embedded project) and a binary version of v4.0.1 (publicly available via the legacy InterNiche website). In those versions, we analyzed the following stack components (see Figure 1): IPv4, TCP, UDP, HTTP, DHCPv4 Client and Server and DNSv4 Client. We performed the analysis by combining manual and automatic procedures, using the following tools:

- The source code version was manually analyzed and fuzzed with libFuzzer.
- The binary version was manually and automatically analyzed by JFrog Security Research, leveraging both static and dynamic proprietary techniques.

INFRA:HALT is the result of a combined effort by Forescout Research Labs and JFrog Security Research. Forescout Research Labs brought to the table the body of knowledge acquired while executing on Project Memoria, while JFrog Security Research provided its platform for automated binary analysis and extensive experience in embedded software security gained from the recent acquisition of Vdoo by JFrog.

Table 2 shows the new vulnerabilities we found. All versions before 4.3 (the latest at the time of research), including NicheLite, are affected. HCC Embedded released patches for the affected versions of NicheStack that are available upon request.

| CVE ID | Vendor ID | Description | Affected Component | Potential Impact | CVSSv3.1 Score |
|---|---|---|---|---|---|
| 2020-25928 | HCCSEC-000010 | The routine for parsing DNS responses does not check the "response data length" field of individual DNS answers, which may cause OOB-R/W. | DNSv4 | RCE | 9.8 |
| 2021-31226 | HCCSEC-000003 | A heap buffer overflow exists in the code that parses the HTTP POST request due to lack of size validation. | HTTP | RCE | 9.1 |
| 2020-25767 | HCCSEC-000007 | The routine for parsing DNS domain names does not check whether a compression pointer points within the bounds of a packet, which leads to OOB-R. | DNSv4 | DoS<br>Infoleak | 7.5 |
| 2020-25927 | HCCSEC-000009 | The routine for parsing DNS responses does not check whether the number of queries/responses specified in the packet header corresponds to the query/response data available in the DNS packet, leading to OOB-R. | DNSv4 | DoS | 8.2 |
| 2021-31227 | HCCSEC-000004 | A heap buffer overflow exists in the code that parses the HTTP POST request due to an incorrect signed integer comparison. | HTTP | DoS | 7.5 |
| 2021-31400 | HCCSEC-000014 | The TCP out of band urgent data processing function would invoke a panic function if the pointer to the end of the out of band urgent data points out of the TCP segment's data. If the panic function had a trap invocation removed, it would result in an infinite loop and therefore a DoS (continuous loop or a device reset). | TCP | DoS | 7.5 |
| 2021-31401 | HCCSEC-000015 | The TCP header processing code doesn't sanitize the length of the IP length (header + data). With a crafted IP packet, an integer overflow would occur whenever the length of the IP data is calculated by subtracting the length of the header from the length of the total IP packet. | TCP | App-dependent | 7.5 |
| 2020-35683 | HCCSEC-000011 | The code that parses ICMP packets relies on an unchecked value of the IP wpayload size (extracted from the IP header) to compute the ICMP checksum. When the IP payload size is set to be smaller than the size of the IP header, the ICMP checksum computation function may read out of bounds. | ICMP | DoS | 7.5 |
| 2020-35684 | HCCSEC-000012 | The code that parses TCP packets relies on an unchecked value of the IP payload size (extracted from the IP header) to compute the length of the TCP payload within the TCP checksum computation function. When the IP payload size is set to be smaller than the size of the IP header, the TCP checksum computation function may read out of bounds. A low-impact write-out-of-bounds is also possible. | TCP | DoS | 7.5 |
| 2020-35685 | HCCSEC-000013 | TCP ISNs are generated in a predictable manner. | TCP | TCP spoofing | 7.5 |
| 2021-27565 | HCCSEC-000017 | Whenever an unknown HTTP request is received, a panic is invoked. | HTTP | DoS | 7.5 |
| 2021-36762 | HCCSEC-000016 | The TFTP packet processing function doesn't ensure that a filename is null-terminated, therefore a subsequent call to strlen() upon the file name might read out of bounds of the protocol packet buffer. | TFTP | DoS | 7.5 |
| 2020-25926 | HCCSEC-000005<br>HCCSEC-000008 | The DNS client does not set sufficiently random transaction IDs. | DNSv4 | DNS cache poisoning | 4 |
| 2021-31228 | HCCSEC-000006 | Attackers can predict the source port of DNS queries to send forged DNS response packets that will be accepted as valid answers to the DNS client's request. | DNSv4 | DNS cache poisoning | 4 |

*Table 2 – Vulnerabilities. Rows are colored according to the CVSS score: yellow for medium or high and red for critical.*

INFRA:HALT exemplifies how all the problems with TCP/IP stacks that we have seen before in Project Memoria can appear in the same product. There are examples of memory corruption issues like AMNESIA:33 (on ICMPv4 and TCPv4, like CVE-2020-35683 and CVE-2020-35684), weak ISN generation like NUMBER:-JACK (CVE-2020-35685), and DNSv4 issues like NAME:WRECK (CVE-2020-25767, CVE-2020-25926, CVE-2020-25927, CVE-2020-25928 and CVE-2021-31228).

## 3. An Attack Scenario Leveraging INFRA:HALT

INFRA:HALT includes remote code execution vulnerabilities that can be exploited to allow attackers to achieve different goals based on their motivations (e.g., infrastructure disruption in case of nation-state sponsored attacks). The technical details of the exploits are discussed in Sections 7 and 8. In this section, we discuss an example of an attack that we implemented in the Forescout Cyber Lab. The attack leverages the DNS-based exploitation detailed in Section 7. The goal of the attacker in this scenario is to disrupt a building's HVAC system, whose controller can be reached by a vulnerable NicheStack device over the network.

The attack scenario is shown in Figure 3, containing the following components:

- **External attacker** (IP address **192.168.85.70**): a malicious actor that leverages a vulnerable device to infiltrate the target network and carry out the attack. The actor is located outside of the local target network and has access to only the subnet 192.168.1.0/24.
- **Device 1** (IP address **192.168.1.21**): a device that runs a vulnerable version of the NicheStack TCP/IP stack. This is the primary target of the attacker since CVE-2020-25928 can be exploited against this device because it sends DNS requests.
- **Device 2** (IP address **192.168.2.14**): a Programmable Logic Controller (PLC) placed in the internal network that controls the operation of a physical device (industrial fan). The PLC is the secondary goal of the attacker since it may contain other vulnerabilities or simply accept unauthenticated commands (as is common in industrial settings). Because the attacker has no direct access to the subnet where the PLC is deployed (192.168.2.0/24), the attacker will attempt to force the vulnerable NicheStack device that has access to this subnet to send a malicious packet to the PLC.

- **Industrial fan** is connected to the PLC, together with a motion sensor; whenever the motion sensor is triggered, the fan starts spinning for several seconds and then halts. This scenario simulates the working conditions of an HVAC (Heating Ventilation Air Conditioning) system, used to control the temperature in mission-critical environments such as data centers or drug storage systems. The attack will result in a Denial-of-Service for the PLC. This will halt the feedback loop between the PLC and the fan so that the fan will remain in its current state permanently, effectively disrupting the process.

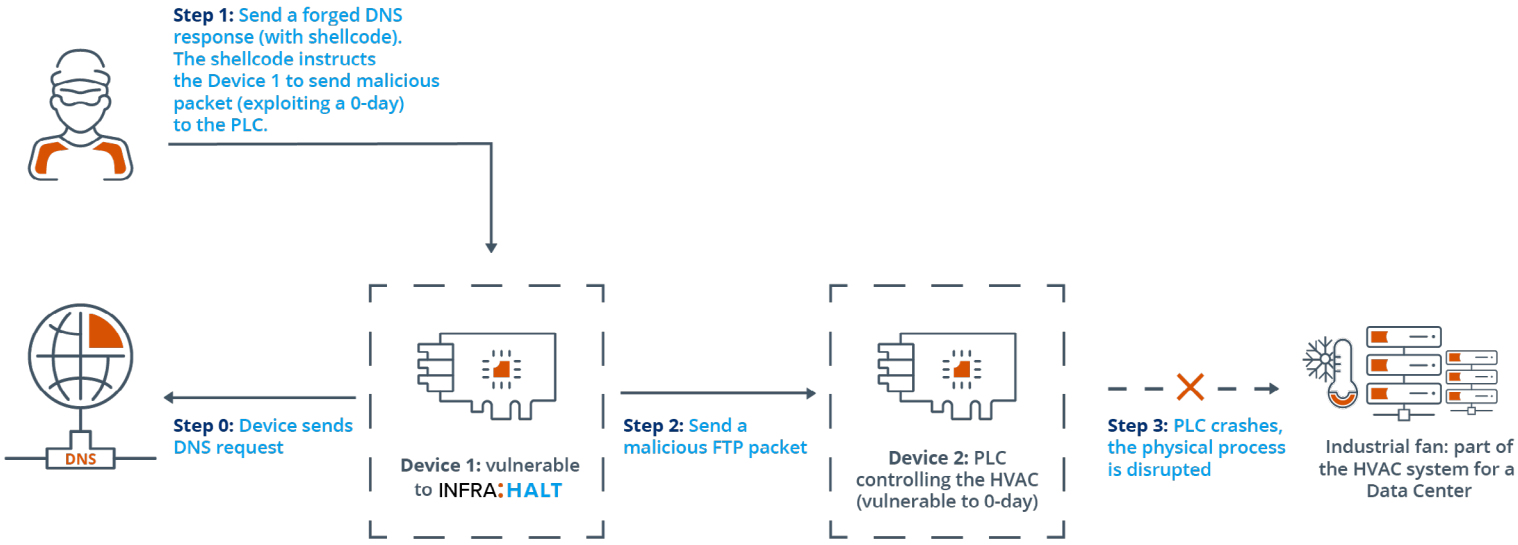


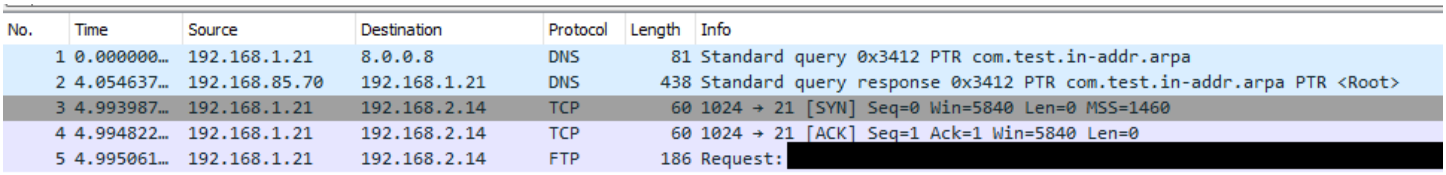

*Figure 3 – Attack scenario on the lab*

The steps of the attack implemented in our lab are as follows:

- **Step 0:** Device 1, vulnerable to INFRA:HALT, sends a DNS request to the DNS server as part of its normal operations.
- **Step 1:** The attacker sends a forged DNS response containing malicious shellcode to Device 1.
- **Step 2:** When Device 1 attempts to parse the DNS response, its logic is hijacked and the attacker gets remote control over it. The device is instructed to establish a TCP connection with Device 2, the internal PLC connected to the HVAC, and to send a malicious FTP packet that exploits a 0-day in this PLC[1].
- **Step 3:** The PLC crashes, forcing the fan control to stop working.

A video showing the effects of the attack on our Cyber Lab is available here.

1 *The 0-day in Device 2 has been discovered as part of our research activities, and it is currently under the responsible disclosure process.*



| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000… | 192.168.1.21 | 8.0.0.8 | DNS | 81 | Standard query 0x3412 PTR com.test.in-addr.arpa |
| 2 | 4.054637… | 192.168.85.70 | 192.168.1.21 | DNS | 438 | Standard query response 0x3412 PTR com.test.in-addr.arpa PTR <Root> |
| 3 | 4.993987… | 192.168.1.21 | 192.168.2.14 | TCP | 60 | 1024 → 21 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 |
| 4 | 4.994822… | 192.168.1.21 | 192.168.2.14 | TCP | 60 | 1024 → 21 [ACK] Seq=1 Ack=1 Win=5840 Len=0 |
| 5 | 4.995061… | 192.168.1.21 | 192.168.2.14 | FTP | 186 | Request: [redacted] |

*Figure 4 – Malicious network communications (exploitation of CVE-2020-25928)*

Figure 4 shows the network capture of the implemented scenario. The malicious network packets are the packets number 2 and 5. We can see Device 1 performs a DNS request (packet number 1) and the attacker immediately responds with a malformed DNS answer that exploits CVE-2020-25928 (packet number 2). Device 1 accepts the malformed DNS answer coming from the attacker, and (as shown by packets number 3 and 4) the shellcode supplied by the attacker is being executed since Device 1 establishes a successful TCP connection with Device 2 (the PLC)[2] . Finally, Device 1 sends a malicious FTP message to Device 2 (packet 5), crashing it.

This scenario could be expanded to a large-scale denial-of-service: attackers could gain full control over exposed devices by exploiting CVE-2020-25928 and then make these devices part of a botnet to carry out a DDoS attack (Distributed Denial of Service) on internal controllers.

The internal controllers exploited could be not only building automation PLCs but also controllers used in manufacturing plants, power generation/transmission/distribution, water treatment and several other critical infrastructure sectors.

2 *Note that the SYN-ACK packet from the TCP Handshake was not captured and thus not shown in Figure 4.*

# 4. Impact

In this section, we try to estimate the impact of INFRA:HALT based on the evidence collected during our research, using three main sources:

- **A legacy InterNiche website listing its main customers.** According to the website, most of the top industrial automation companies in the world, such as Emerson, Honeywell, Mitsubishi Electric, Rockwell Automation, Schneider Electric and Siemens, use the stack. Besides those, the website mentions a total **of almost 200 device vendors**.
- **Shodan Queries.** Shodan is a search engine that allows users to look for devices connected to the Internet.

We queried Shodan, looking for devices showing some evidence (e.g., application-layer banners) indicating the use of NicheStack. As shown in Figure 5, with a query executed on 08/Mar/2021, we found more than 6,400 instances of devices running NicheStack (using the simple query "InterNiche"). Of those devices, the large majority (6360) run an HTTP server (query "InterNiche Technologies Webserver"), while the others ran mostly FTP ("Welcome to InterNiche embFtp server"), SSH ("SSH-2.0-InterNicheSSHServer (c)InterNiche") or Telnet ("Welcome to InterNiche Telnet Server") servers.


*Figure 5 – Results for "InterNiche" on Shodan*


*Figure 6 – Results for "InterNiche Technologies Webserver" on Shodan*

- Forescout Device Cloud. Forescout Device Cloud is a repository of information of 13+ million devices monitored by Forescout appliances. We queried it for similar banners as Shodan, as well as other information, based on DHCP signatures, for instance.

We found more than 2,500 device instances from 21 vendors. The most affected customer industry vertical is Process Manufacturing, followed by Retail and Discrete Manufacturing.

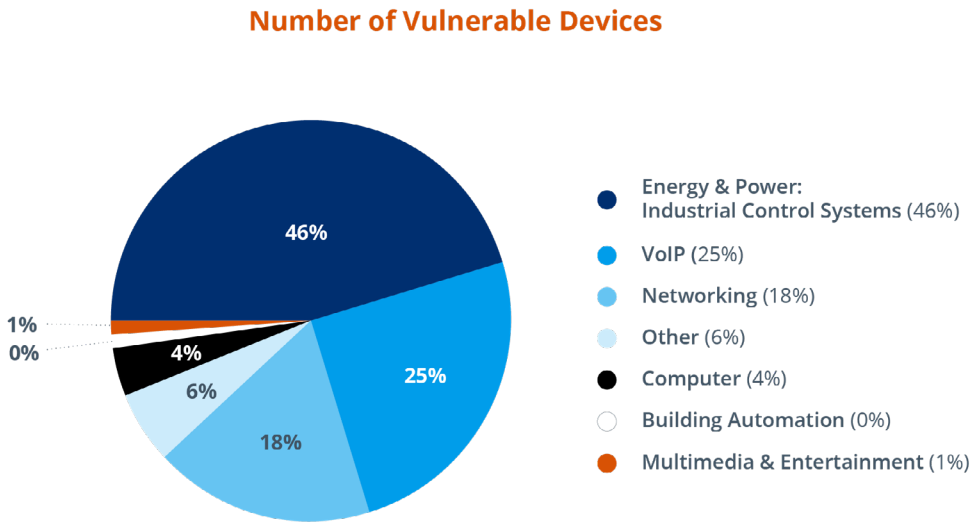



*Figure 7 – Device Functions running NicheStack (source: Forescout Device Cloud)*

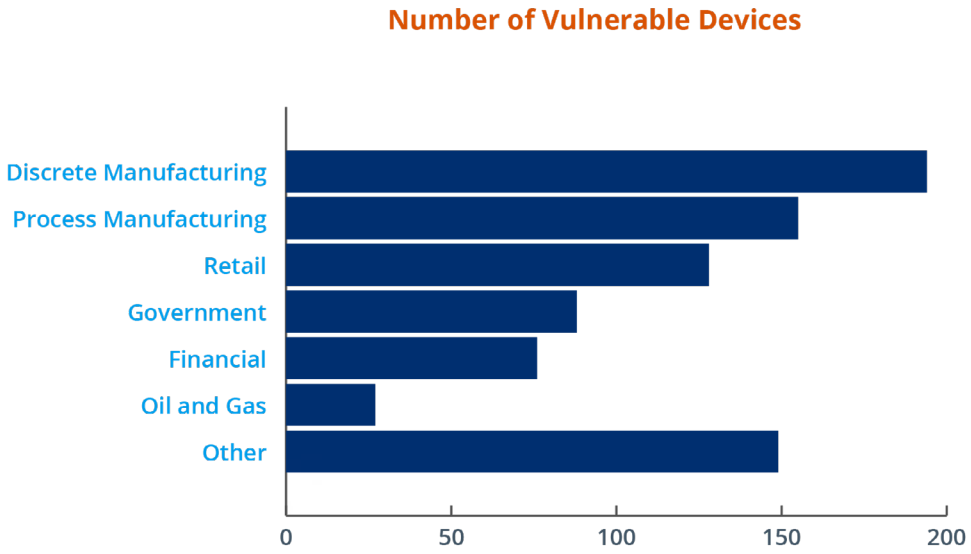



*Figure 8 – Devices running NicheStack in each vertical*

# 5. Mitigation Recommendations

Complete protection against INFRA:HALT requires patching devices running the vulnerable versions of NicheStack. HCC Embedded has released its official patches and device vendors using this software should provide their own updates to customers. Below, we discuss mitigation strategies for network operators, device vendors and the wider cybersecurity community.

## 5.1. For network operators

Given that patching OT devices is notoriously difficult due to their mission-critical nature, we recommend the following mitigation strategy:

- **Discover and inventory devices running NicheStack.** Forescout Research Labs has released an open-source script that uses active fingerprinting to detect devices running NicheStack. The script is updated constantly with new signatures to follow the latest development of our research.
- **Enforce segmentation controls and proper network hygiene** to mitigate the risk from vulnerable devices. Restrict external communication paths and isolate or contain vulnerable devices in zones as a mitigating control if they cannot be patched or until they can be patched.
- **Monitor progressive patches released by affected device vendors** and devise a remediation plan for your vulnerable asset inventory, balancing business risk and business continuity requirements.
- **Monitor all network traffic for malicious packets** that try to exploit known vulnerabilities or possible 0-days. Anomalous and malformed traffic should be blocked, or at least alert its presence to network operators.

Table 3 provides recommended mitigations for each vulnerability.

| CVE | Affected Component | Mitigation Recommendation |
|---|---|---|
| 2020-25928<br>2020-25767<br>2020-25927<br>2021-31228<br>2020-25926 | DNSv4 client | Disable the DNSv4 client if not needed, or block DNSv4 traffic. Because there are several vulnerabilities that facilitate DNS spoofing attacks, using internal DNS servers may not be sufficient (attackers may be able to hijack the request-response matching). |
| 2021-27565<br>2021-31226<br>2021-31227 | HTTP | Disable HTTP if not needed, or whitelist HTTP connections. |
| 2021-31400<br>2021-31401<br>2020-35684<br>2020-35685 | TCP | For CVE-2021-31400, CVE-2021-31401, and CVE-2020-35684, we recommend monitoring traffic for malformed IPv4/TCP packets and blocking them (e.g., having a vulnerable device behind a properly configured firewall should be sufficient).<br>For CVE-2020-35685, we suggest using recommendations we outlined in our NUMBER:JACK report whenever it is feasible. |
| 2020-35683 | ICMPv4 | Monitor traffic for malformed ICPMv4 packets and block them. |

*Table 3 – Mitigation recommendations for specific vulnerabilities*

### 5.2. For device vendors

**Use exploit mitigations.** No stack canary or ASLR mitigations were built into the SAM4E binary mentioned in Sections 3 and 7. This is common in embedded systems even nowadays. These mitigations would make exploitation much more difficult and, in most cases, impossible without the use of other information disclosure vulnerabilities. Device vendors should ship their toolchains configured to utilize these mitigation techniques by default.

**Use code-auditing tools such as binary analysis, source analysis and fuzzing.** Some of the vulnerabilities in INFRA:HALT can be found with modern analysis tools in a completely automated manner. Device vendors should employ source code analysis, binary analysis and dynamic analysis (fuzzing), as each technique has advantages and disadvantages for finding security issues.

**When implementing well-known protocols, use well-known security techniques when possible.** CVE-2020-35685 could have been avoided by following the proposed ISN generation algorithm in RFC 6528. In many well-known protocols, the security-related questions have been publicly and thoroughly answered in RFCs and other documentation. We recommend looking for these solutions before trying to implement a new algorithm.

### 5.3. For the community

As with every supply chain vulnerability, identifying all impacted devices might require months or even years, leaving vulnerable assets exposed for a long time. In the case of AMNESIA:33, publicly disclosed in December 2020, updates regarding affected devices have still been published in May 2021, five months after the initial publication (and eight months after the initial notification to vendors). This is because identifying product lines that might include a vulnerable component, verifying if any product in the line is affected and providing a fix are **lengthy, manual and difficult processes**.

To facilitate this process, Forescout and JFrog shared the details of their findings about potentially affected vendors with CERT/CC, ICS-CERT and BSI, which coordinated the disclosure with these vendors. The Forescout Device Cloud was used to identify devices that show some evidence (e.g., HTTP banners, Nmap fingerprints, etc.) of the presence of a vulnerable component. This information, when shared with the appropriate parties, can make it easier to identify vendors that must be notified during responsible disclosure. In INFRA:HALT, we identified nearly 200 vendors possibly affected. Not all of them will be confirmed vulnerable, since evidence and fingerprints might lead to false positives.

The cybersecurity community (researchers, security vendors, device manufacturers and other actors) should work cooperatively to find better (and possibly automated) ways of identifying software components of a device. An example of such a community effort is led by the National Telecommunications and Information Administration (NTIA) and has the goal of creating a common machine-readable exchange format for Software Bills of Materials (SBOM). That SBOM format would uniquely and effectively identify software components. Having access to the list of software components that comprise a software or hardware solution would allow to clearly establish if a device is affected by a certain vulnerability by simply 'reading' its SBOM.

This initiative is complemented by the OASIS Common Security Advisory Framework (CSAF) and the Vulnerability Exploitability Exchange (VEX), which both make SBOMs more usable for defenders. CSAF is a way for researchers, vendors and coordinators to provide security advisories in a machine-readable way. This aids in the efforts of end users and downstream vendors to react faster on published patch and remediation information as it becomes automatable. VEX, on the other hand, allows device vendors, software providers and others to explicitly state in a machine-readable way when they are not affected by a particular vulnerability, thus reducing the false positives that are generated by fingerprints. Both, when combined with an SBOM, allow organizations to better understand the implications of vulnerabilities on their products or networks and take better risk-based decisions.

TECHNICAL DIVE-IN

# 6. Technical Dive-In #1: An Example of Automated Vulnerability Discovery

The automated binary analysis that found many of the vulnerabilities in INFRA:HALT was performed on a publicly available demo version of NicheStack for the SAM4E microcontroller. The demo binary is a standard non-stripped ELF image.

JFrog's platform for security analysis automatically identifies user input points that are notoriously linked to possible vulnerabilities.

As an example, to detect the vulnerable data path in CVE-2021-31228 (an HTTP server denial-of-service), the JFrog static analyzer framework started by identifying a potential point of user input, the **atol()**[3] call in function **ht_readmsg()** shown in Figure 9.

```
contentlen = atol(pcVar6 + 0xf);
hp->contentlen = contentlen;
if ((hp->rxsize - (int)(pcVar4 + -(int)pcVar8) < hp->contentlen) &&
    (pcVar5 = stristr(pcVar5,"multipart/form-data"), pcVar5 == (char *)0x0)) {
  return;
}
```

*Figure 9 – User input automatically detected*

This is automatically regarded as a possible user input since string-to-integer conversion functions are mainly used to convert textual user input data to integers that the program can work with.

Also, this is cross-referenced with well-known protocol strings in the function's proximity. In this case, the HTTP-related strings give the system higher confidence that this is indeed user input coming from the network.

3 *Since the ELF was non-stripped in this case, "atol" was identified automatically by name. However, even in a stripped ELF, it can be automatically identified easily by emulating it and using test-case divination analysis.*

TECHNICAL DIVE-IN

In this vulnerability, the issue is that *contentlen* may be provided by the attacker as a negative number, which will help evade any signed comparisons while still causing huge data copies when treated as an unsigned integer. In this case, *contentlen* is stored in the **hp** struct, which is then tracked in the program until the function **wbs_post**, where it is used to initialize the total and remaining length, as shown in Figure 10.

```
phVar6->totlen = hp->contentlen;
phVar6->remain_len = hp->contentlen;
```

*Figure 10 – contentlen used in the wbs_post function*

This struct is tracked even further until it reaches one of the potential "sink" functions, in this case, a **memcpy()** call in **getbndsrch**, as shown in Figure 11.

Here, *len* is a value derived from *remain_len*, which in turn is derived from *contentlen* (as shown in Figure 10).

Since *len* can be negative and the comparison is signed, the "**if**" branch will be taken, and a huge copy operation will occur, crashing the device due to invalid memory write access.

```
char * getbndsrch(htupload *htup,char *cp,int len,int *err)

{
  int iVar1;
  int i;

  if (len < htup->boundarylen) {
    memcpy(htup->pbuf,cp,len);
    htup->curlen = len;
  }
```

*Figure 11 – memcpy() sink*

TECHNICAL DIVE-IN

# 7. Technical Dive-In #2: DNS-Based Exploitation (CVE-2020-25928)

There are two vulnerabilities in INFRA:HALT that allow for Remote Code Execution: CVE-2020-25928 and CVE-2021-31226. These vulnerabilities enable attackers to remotely take over the target devices. In this section, we will focus on CVE-2020-25928: a missing size check when handling DNS responses, for the length of the response data. The missing check leads to an exploitable heap buffer overflow. In Section 8, we discuss the exploitation of CVE-2020-31226.

At a high level, to trigger CVE-2020-25928, an attacker sends a crafted DNS packet as a response to a DNS query from the vulnerable device. This is easy to achieve because the DNS TXID and UDP source port can be guessed due to CVE-2020-25926 and CVE-2021-31228, respectively, and the affected DNS client implementation does not validate the source IP address of the response packet (so the attacker does not even need to know the address of the real DNS server). Therefore, a man-in-the-middle is not needed to exploit CVE-2020-25928. A passive sniff of a DNS query at some point in time will greatly shorten the number of options for the DNS TXID and UDP source port, allowing the vulnerability to be exploited with a handful of response packets (less noise on the wire), although a brute force is quite possible even without an initial passive sniff.

The crafted packet contains malicious code ("shellcode") that hijacks the logic of a vulnerable NicheStack device attempting to parse it and instructs the device to execute a malicious action. In the case of the proof-of-concept below, the device establishes a TCP connection with another networked device, which can be used for further exploitation.

The vulnerability is detailed in Section 7.1. The actual exploitation is detailed in Section 7.2, and the shellcode is detailed in Section 7.3.

**Important note on exploitability:** *Some of the technical details of the exploitation are specific to the physical device being exploited, including the presence of specific components of the affected TCP/IP stack and the absence of exploit mitigations. The details discussed below are specific to the physical target we used: an ATSAM4E development board running the public binary demo mentioned in Section 3. However, this exploit can be generalized to other targets.*

TECHNICAL DIVE-IN

## 7.1. Vulnerability details

CVE-2020-25928 occurs when individual resource records (RRs) of a DNS response packet are processed. Figure 12 shows a pseudocode excerpt from the **dns_upcall()** function that is called whenever a DNS response packet is received. At line 1, a byte pointer c**p** is initialized, pointing to the second byte of the DNS header of the received response packet (the **dnshdr** structure). The **for()** loop (lines 5–40) iterates over the available DNS records and extracts the corresponding record fields. The variable records hold the number of records it has extracted from the DNS header **dnshdr** earlier.

```
0: uint8_t *cp;
1: cp = (uint8_t *)&dnshdr[1];
2: dns_entry->alist[0] = 0;
3: dns_entry->ipaddrs = 0;
4:
5: for ( i = 0; records > i; ++i )
6: {
7:     // ...
8:     cp = getoffset(cp, (unsigned __int8 *)dnshdr, &offset);
9:     cp = getshort(cp, &type);
10:    cp = getshort(cp, &netclass);
11:    // ...
12:    cp = getlong(cp, &ttl);
13:    cp = getshort(cp, &rdlength);
14:    switch(type)
15:    {
16:        // ...
17:        case 0xCu:
18:            if ( type == 1 && rdlength != 4 )
19:              err = 7;
20:            if ( !err )
21:            {
22:              ++dnsc_good;
23:              if ( queries + answers <= i )
24:              {
25:                if ( nameoffset == offset )
26:                {
27:                  dnc_set_answer(dns_entry, type, cp, rdlength);
28:                }
29:                // ...
30:              }
31:              // ...
32:            }
33:            break;
34:        // ...
35:        case 0xFu:
36:            break;
37:        // ...
38:    }
39:    //...
40: }
```

*Figure 12 – An excerpt from the dns_upcall() function (CVE-2020-25928)*

While individual records are parsed, the pointer **cp** iterates over various fields of every available resource record at lines 8–13. The functions **getoffset()** and **getshort()** are used to retrieve various fields of the record: type corresponds to the DNS record type; **netclass** corresponds to the DNS record class (e.g., "IN" for "Internet"); **ttl** is the Time-to-Live value; and **rdlength** is a two-byte field that specifies the length of the response data that follows.

When the values of these fields are extracted, the function **dnc_set_answer()** is called to retrieve the response data and write it into **dns_entry** (this is a pointer to the **dns_querys** structure shown in Figure 13).

```
typedef struct _dns_querys {
  ...
  char dns_names[256];
  char ptr_name[128];      // +128
  void *auths_ip;          // +4
  void *alist[3];          // +12
  int protocol;            // +4
  hostent he;              // +32
  int type;                // +4
  uint8_t opcode[3];       // +3
  void *h_TXT_list;        // +4
  int h_TXT_listCount;     // +4
} dns_querys;              // distance from start of ptr_name to end of struct is 195 bytes
```

*Figure 13 – The **dns_querys** structure (CVE-2020-25928)*

TECHNICAL DIVE-IN

**dnc_set_answer()** accepts a pointer to the **dns_querys** structure (shown in Figure 13), the DNS record type (**type**), the current byte pointer **cp**, and the length of the DNS response data (here, **rdlen**). If the current record is a domain name pointer record (**type** is set to *12*, or *0x0c* in hexadecimal), the function will call **memcpy()** to write the response data into **dns_entry->ptr_name**, as per Figure 14. Here, the first argument of **memcpy()** is the destination, the second argument is the source (**cp + 1** points to the beginning of the response data), and the third argument is the amount of bytes to be copied.

```
1: void dnc_set_answer(dns_querys *dns_entry, unshort type, uint8_t *cp, int rdlen)
2: {
3:   // ...
4:   switch ( type )
5:   {
6:     // ...
7:     case 0x0c:
8:       memcpy(dns_entry->ptr_name, cp + 1, rdlen - 1);
9:       // ...
10:      break;
11:    // ...
12:  }
13: }
```

*Figure 14 – An excerpt from the **dnc_set_answer()** function*

Figure 13 shows that the field **ptr_name** has a fixed size (128 bytes) and that the resource data length value is never checked along the way. The size of the memory copy operation is controlled by potential attackers, and arbitrary resource data length value can be specified directly in a network packet, making the memory copy write past the buffer **ptr_name**, up to 65,535 bytes (the size limit for the short-sized **rdlen**).

The memory for each **dns_entry** is allocated in the heap (using **malloc()**, this code is omitted for brevity), and therefore it is a classic heap overflow vulnerability. In the next section, we discuss how this vulnerability can be exploited to achieve Remote Code Execution.

### 7.2. Exploiting the vulnerability

To exploit CVE-2020-25928, an attacker must forge a DNS response packet that includes an RR with the structure shown in Figure 15. This RR may or may not specify the same domain name that has been requested by the vulnerable device (e.g., “test.com”); it must be of a domain pointer record (type set to 0x000c), the DNS record class code must be set to “Internet” (0x0001), the resource data length must be set to a sufficiently large value to cause a buffer overflow and to ensure that the shellcode is being written entirely (we provide the value of 401 bytes). Finally, the desired shellcode must be in place of the resource data.

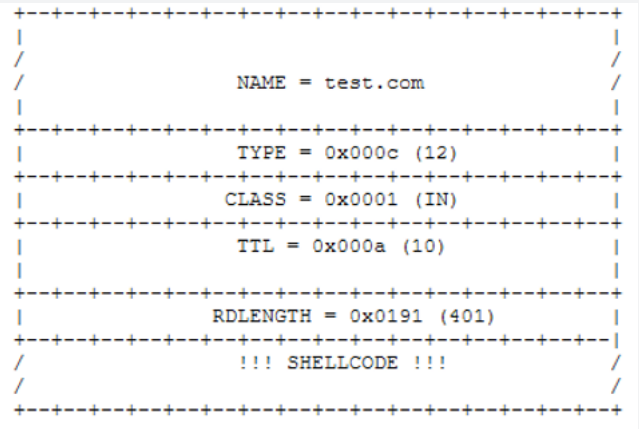



*Figure 15 – The RR structure of a forged DNS response packet (CVE-2020-25928)*

TECHNICAL DIVE-IN

Exploiting heap overflows involves either corrupting heap metadata (the data structures created and managed by the memory allocator) or corrupting the program data stored on the heap. For this PoC, we chose the former approach. Therefore, the shellcode is specific to the memory allocator used in the vulnerable firmware, the hardware architecture and the available functionality within the vulnerable firmware.

Understanding how the target memory allocator works is crucial to achieving RCE via heap metadata overflow. Thus, we sketch the specifics of the memory allocator relevant to the firmware and the architecture of our target below.

### The memory allocator

Figure 16 shows how the **dns_entry** structure is allocated. The memory allocator used in NicheStack is very similar to the memory allocator of **newlib**[4] (the **malloc()**[5] function is called by **npalloc()**). Therefore, we will use newlib to illustrate the inner workings of the memory allocator used by NicheStack.

```
2   dns_querys * dnc_new(void)
3
4   {
5     dns_querys *pdVar1;
6     dns_querys *dns_entry;
7
8     pdVar1 = (dns_querys *)npalloc(0x210);
9     if (pdVar1 != (dns_querys *)0x0) {
10      pdVar1->rcode = 0xff;
11      (pdVar1->he).h_aliases = pdVar1->alist;
12      pdVar1->next = dns_qs;
13      dns_qs = pdVar1;
14      }
15      return pdVar1;
16  }
```

*Figure 16 – **dns_entry** structure allocation*

```
struct malloc_chunk
{
  INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if free). */
  INTERNAL_SIZE_T size;      /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;   /* Forward pointer, points to the next free chunk in
                                the dobly linked list -- used only if free */

  struct malloc_chunk* bk;   /* Backward pointer, points to the previous free
                                chunk in in the doubly linked list -- used only
                                if free */
};
```

*Figure 17 – The **malloc_chunk** structure in **newlib***

The following specifics of the memory allocator are important for our discussion:

- **Memory chunks**, which are allocated/free areas of memory.
- **Bins**, which are double linked lists of free chunks. There may be several kinds of bins, depending on the sizes of the chunks they can hold.

4 https://sourceware.org/newlib/

5 https://chromium.googlesource.com/native_client/nacl-newlib/+/refs/heads/main/newlib/libc/stdlib/mallocr.c
*(Note that the code might not be exactly the same, but the crucial parts are similar enough.)*

TECHNICAL DIVE-IN

Figure 17 shows the data structure **(malloc_chunk)** that holds individual free chunks. It has the following fields:

- **prev_size** – the size of the previous chunk.
- **size** – the size of the current chunk. If the least significant bit of this value is unset, it means that the previous chunk is free and can be used for allocation or for merging with other chunks.
- **fd** – the forward pointer, which points to the next free chunk in the double linked list, used only if there is a free chunk after the current one in the free list.
- **bk** – the backward pointer, which points to the previous free chunk in the double linked list, used only if there is a free chunk before the current one in the free list.

The **top bin** (or **top chunk**) is a region of heap memory that holds the topmost free chunk. It is a single chunk of contiguous free memory, and it is also the largest free chunk available for the memory allocator. The **top bin** is used when there are no other bins to hold free chunks of the appropriate size.

The allocation of a chunk is performed by calling **malloc()**. The memory allocation algorithm behind the scenes performs the following steps (the description is simplified):

- If there is a chunk of memory that has been just freed, and it is large enough to accommodate the request, the memory allocator will use it.
- If not, and there is space at the top of the heap (top bin), the memory allocator will create a new chunk out of this memory region and use it.
- If the top bin is too small to accommodate the request, the memory allocator will instruct the kernel to add new memory at the end of the heap. It will then consolidate the new memory region (new top bin) with the old contiguous free chunk (old top bin). This space will be used for allocation.
- Otherwise, malloc() fails and returns NULL.

TECHNICAL DIVE-IN

## Overflowing the heap

Figure 18 illustrates the state of the heap before and after we overflow the **dns_entry->ptr_name** buffer. In our case, **dns_entry** is allocated right above the **top bin** and is adjacent to it. We choose the length of the payload (forged DNS resource record data) to overflow the **ptr_name** buffer, overwrite the remaining fields of the **dns_entry** structure, and overwrite the metadata of the current **top bin**.

We choose the payload in such a way that there is now a **fake chunk** within the memory chunk allocated for the **dns_entry** structure, and the metadata of the **top bin** is modified.

Whenever a new memory allocation through **malloc()** takes place, the memory allocator will attempt to use the **top bin**.

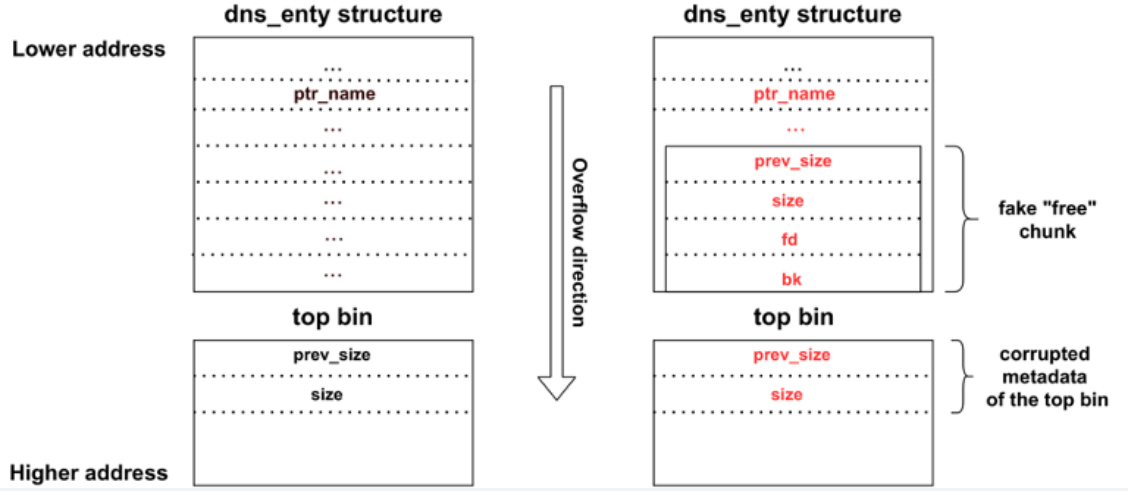



*Figure 18 – The state of the heap after buffer overflow (CVE-2020-25928)*

We can set the **size** field of the **top bin** through the overflow, such that the memory allocator will deem that there is not enough space, and it will attempt to extend the **top bin**. After this extension takes place, the memory allocator will free the **top bin**. It will also attempt to consolidate the freed **top bin** with any other adjacent chunks that are also free. This is where the **fake chunk** comes into play: we unset the least significant bit of the **size** field of the **top chunk**, indicating that the fake chunk is "free." By carefully placing addresses within this **fake chunk** (forward and backward pointer fields), attackers can achieve arbitrary memory writes (e.g., write-what-where[6] ), hijack the control flow of a vulnerable program, and execute arbitrary code. These are the elements of the classic heap exploitation technique sometimes known as "unlink() technique"[7].

6 *https://cwe.mitre.org/data/definitions/123.html*
7 *http://phrack.org/issues/57/8.html*

TECHNICAL DIVE-IN

To achieve our goal, we need to make sure that only the **top bin** is available for the allocation of the **dns_entry** and the next memory allocation. As the heap memory is volatile, we cannot always meet this condition without shaping the heap in a certain way. These techniques are out of scope of this proof-of-concept; therefore, we always rely on the initial state of the heap after we reset the target device (which is always predictable in our case).

```
1: //...
2:   tk_res_lock(0x15,0x7fffffff);
3:   ho = gethostbyname2(cp,af);
4:   if (ho == (hostent *)0x0) {
5:     gio_printf(gio,"nslookup failed.\n");
6:   }
7: // ...
8:
9: int gio_printf( /* ... */) {
10: // ...
11:   buf_size = 0x9c;
12:   uStack8 = in_r2;
13:   uStack4 = in_r3;
14:   sVar1 = strlen(format);
15:   if (0x9b < (int)sVar1) {
16:     buf_size = sVar1 + 0x9c;
17:   }
18:   str = (char *)npalloc(buf_size);
19:   if (str == (char *)0x0) {
20:     ret_value = -0x14;
21:   }
22:   else {
23:     vsprintf(str,format,(va_list)&uStack8);
24:     sVar1 = strlen(str);
25:     if (buf_size <= (int)sVar1) {
26:                     /* WARNING: Subroutine does not return */
27:       panic("gio_printf:Buffer overflow");
28:     }
29:     if (gio == (GIO *)0x0) {
30:       sVar1 = strlen(str);
31:       ret_value = gio_out((GIO *)0x0,str,sVar1);
32:     }
33:     else {
34:       uVar2 = gio->flags;
35:       gio->flags = gio->flags | 0x20;
36:       sVar1 = strlen(str);
37:       ret_value = gio_out(gio,str,sVar1);
38:       gio->flags = uVar2;
39:     }
40:     npfree(str);
41:   }
42:   return ret_value;
43: }
```

*Figure 19 – gio_printf() is called when the DNS lookup fails*

When crafting the value for **size** of the **top bin**, we must ensure that the following conditions hold:

1. The least significant bit of **size** must be set to zero. This indicates that the **previous (fake) chunk** is free.
2. The value of **size** must be bigger than the **minimal possible chunk size** (16 bytes).
3. Let **new_size** be the size of the memory (masked by the malloc_align_mask) that will be allocated during the next **malloc()** call that occurs after we create the **fake chunk** and modify the metadata of the **top bin** with overflow. The following condition must hold: **size < new_size**.

In particular, the last condition above means that we must carefully choose the value of size with respect to the next **malloc()** call that happens within the execution path that we wish to hijack. To satisfy the above conditions, we have chosen the value of *0x000000a2*.

Figure 19 shows the function call that we aim to hijack: Because we have sent a malformed DNS response, DNS lookup will fail, and the function **gio_printf()** will be called to log the corresponding error. When this function is called, a string buffer is allocated on the heap, line 18. Because this memory allocation should happen in the **top bin**, malloc() will create a **new top bin** and will then free the old top bin, consolidating it with the **fake "free" chunk**.

TECHNICAL DIVE-IN

```
#define unlink(P, BK, FD) \
{                         \
  BK = P->bk;             \
  FD = P->fd;             \
  FD->bk = BK;            \
  BK->fd = FD;            \
}                         \
```

*Figure 20 – The unlink macro*

To join the freed **top bin** and the **fake "free" chunk**, the memory allocator will use the **unlink** macro (shown in Figure 20). This macro simply removes a chunk node from the double linked list of free chunks. Because we have unset the least significant byte of the size field of the top bin, the memory allocator will deem that the fake chunk we have inserted before is free, and it will attempt to unlink this chunk.

Let P be the pointer to **fake chunk** that we control through the buffer overflow (**BK** and **FD** are just some temporary pointers in the stack). Considering that the vulnerable NicheStack device we used for our exploit has a 32-bit CPU, the **unlink** macro boils down to the following two operations:

***(P->fd + 12) = P->bk**

***(P->bk + 8) = P->fd**

In layman's terms, it means that the memory contents at whatever address we place into the forward pointer of the **fake chunk** plus 12 bytes will be overwritten with whatever address we place into the backward pointer of the **fake chunk**. As an undesired side effect, the contents of the memory at the address that we place into **P->bk** plus 8 bytes will be overwritten with **P->fd**.

The memory allocator in the analyzed versions of NicheStack does not include security checks such as "Safe-Unlinking"[8] (in contrast with, e.g., newer versions of **glibc**[9]). Because of this, and since we have full control over the pointers **P->bk** and **P->fd** through the buffer overflow, the **unlink** macro allows us to achieve arbitrary memory writes. We use this to hijack the control flow from **gio_printf()** as follows:

1. In the overflow payload, we specify the value of **P->fd** to be the stack address that holds the return address for the **gio_printf()** call, minus 12 bytes. For example, if this address is 0x20012c18, we should put 0x20012c0c.

2. In place of **P->bk** (the next address within **fake chunk**), we put the address at which our shellcode begins (e.g., *0x20014b4d)*.

3. The undesired side effect of **unlink** will overwrite the contents of "**P->bk+8**" with **P->fd**, slightly corrupting our shellcode. To alleviate this, we must begin the shellcode with a "jump" instruction that will continue the shellcode execution after the corrupted part.

The result of these manipulations will overwrite the return address of the **gio_printf()** stack frame so that it will not return to its original callee, but instead, our shellcode will be executed.

8 https://research.checkpoint.com/2020/safe-linking-eliminating-a-20-year-old-malloc-exploit-primitive/

9 https://www.gnu.org/software/libc/

TECHNICAL DIVE-IN

### 7.3. The shellcode

The goal of our shellcode is to force the vulnerable device to communicate with other devices over the network. To do so, we use the available network API in NicheStack, namely the functions **t_socket()**, **t_connect()**, **t_send()**, and **t_sendto()**, which can be used for establishing a TCP connection with another network endpoint and for sending data to it over the TCP protocol.

We structure the shellcode in the following way:

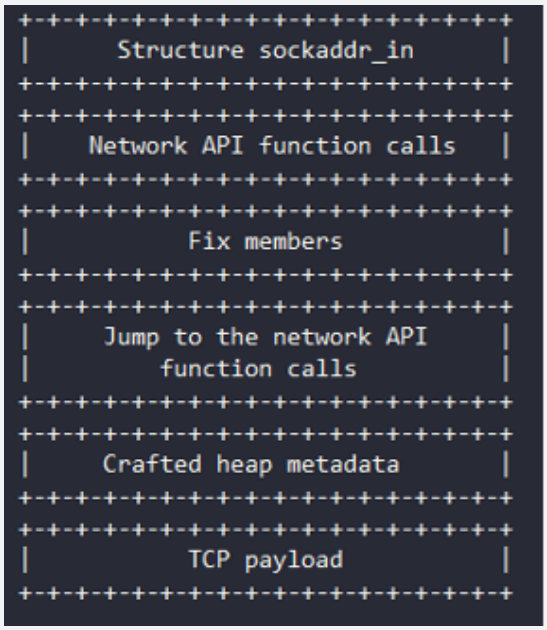



*Figure 21 – Shellcode structure*

We first include bytes for initializing the **sockaddr_in** structure that will be passed into the **t_connect()** function: the fields of this structure contain necessary parameters for establishing a TCP connection (e.g., target IP address and port).

Next, we include the assembly code that executes the above functions to establish a TCP connection and send a TCP data packet to other devices (**t_socket() -> t_connect() -> t_send() -> t_socketclose()**). Since we do not implement any persistence on the target, we include an additional call to **t_reset()** that resets the NicheStack device after the attack is performed.

Next, we attach a byte sequence that ensures that the overwritten members of the **dns_entry** structure will not cause any trouble by crashing the memory allocator unexpectedly ("Fix members"). After this comes the first instruction that is executed after hijacking the return address: We include a jump assembly instruction that will move the current instruction pointer to the first network API function call within the shellcode (e.g., "t_socket()" call). This jump instruction is added to overcome the undesired effect of the **unlink** macro, which will corrupt part of our shellcode, so we can keep the actual shellcode and the shellcode pointer separate in this way (see Section 7.2). Finally, we include the crafted heap metadata that includes the fake free chunk and the fake top bin (see Section 7.2), as well as the data that will be sent to the other network device over the TCP protocol once we hijack the control flow of the NicheStack device ("TCP payload").

TECHNICAL DIVE-IN

# 8. Technical Dive-In #3: HTTP-Based Exploitation (CVE-2021-31226)

Alongside the DNS-based exploitation that was explained thoroughly in the Section 7, we also wrote a preliminary PoC for the HTTP-based RCE (CVE-2021-31226) using similar techniques. Exploitation via HTTP is a good option when the InterNiche HTTP server is enabled on the victim device. It has the benefit of not requiring a DNS request, since the exploiting packet is sent directly to the HTTP server.

Note that the PoC was tested on the same physical hardware that was mentioned in Section 7, so the same exploitability caveats apply in this section as well.

## 8.1. Vulnerability details

CVE-2021-31226 occurs during the parsing of the HTTP Request URI field in the function **ht_readmsg()**. After making sure the packet has a valid "Content-Length" header value, the parsing logic gets the pointer to the request URI (**requri**) by calling **ht_nextarg()** on the HTTP request's buffer and stores this pointer in the **header_struct->fi->requri**. For clarification, the request URI refers to the part highlighted in red of the entire URI (path + query string):

**https://example.org/path/to/file?param=42#fragment**

The code shown in Figure 22 goes over the request URI searching for a "?" character that will terminate **requri**. It also puts a null terminator when encountering a character out of the expected range, which is > 0x20 ASCII.

```
79          phVar9 = hp->fi;
80          pcVar4 = ht_nextarg(hp->rxbuf);
81          phVar9->requri = pcVar4;
82          cp = phVar9->requri;
83          while (0x20 < (byte)*cp) {
84            if (*cp == '?') {
85              *cp = '\0';
86              iVar3 = wbs_mkform(hp,cp + 1,0);
87              if (iVar3 != 0) {
88                ht_kill(hp);
89                return;
90              }
91              break;
92            }
93            cp = cp + 1;
94          }
95          *cp = '\0';
```

*Figure 22 – ht_readmsg() excerpt*

Eventually, the HTTP packet is queued to further processing by the **wbs_post()** function shown in Figure 23. Inside **wbs_post()**, the code looks for the "Content-Type:" string, possibly followed by spaces with a value of "multipart/form-data". Then, **header_struct->upload** gets a fixed-size heap allocation of 0xEC bytes (line 37).

TECHNICAL DIVE-IN

```
36      if (hp->upload == (htupload *)0x0) {
37        phVar6 = (htupload *)npalloc(0xec);
38        hp->upload = phVar6;
39        if (hp->upload == (htupload *)0x0) {
40          ht_senderr(hp,500,htout_of_mem);
41          return -1;
42        }
43      }
44      phVar6 = hp->upload;
45      phVar6->totlen = hp->contentlen;
46      phVar6->remain_len = hp->contentlen;
47      if ((*hp->fi->requri == '/') && (hp->fi->requri[1] != '\0')) {
48        strcpy(phVar6->cgifname,hp->fi->requri);
49      }
```

*Figure 23 – wbs_post() excerpt*

Upon successful allocation, there is a check in line 47 that makes sure **requri** starts with '/' and holds at least one more character. Then, in line 48, the vulnerable **strcpy** is called, copying the **requri** string (whose size was never checked) to the **header_struct->upload->cgifname** field, which is at offset 0xB8 inside the 0xEC allocated **header_struct->upload** buffer. **This means that a requri string of more than 52 bytes would cause a heap overflow.**

### 8.2. Exploiting the vulnerability

The HTTP-based scenario can be exploited in a similar way as the DNS-based scenario. In the **wbs_post()** function, there is a reachable **npalloc()** call shortly after the **strcpy()** call that causes the overflow, as shown in Figure 24.

When setting up the heap similarly to Figure 18, the **npalloc()** call will cause an unlinking operation, which leads to an attacker-controlled arbitrary overwrite.

```
if ((*hp->fi->requri == '/') && (hp->fi->requri[1] != '\0')) {
                /* HEAP OVERFLOW HERE */
  strcpy(phVar6->cgifname,hp->fi->requri);
}
pcVar3 = stristr(pcVar8,"boundary=");
if (pcVar3 == (char *)0x0) {
  ht_senderr(hp,400,"Boundary not found");
  return -1;
}
phVar6->boundary[0] = '-';
phVar6->boundary[1] = '-';
tmp = phVar6->boundary + 2;
boundarylen = 2;
cp = pcVar3 + 9;
if (*cp == '\"') {
  cp = pcVar3 + 10;
  bVar1 = true;
}
while (((__ctype_ptr__[(byte)*cp + 1] & 8U) == 0 && (boundaryle
  *tmp = *cp;
  tmp = tmp + 1;
  cp = cp + 1;
  boundarylen = boundarylen + 1;
}
if (bVar1) {
  boundarylen = boundarylen + -1;
  tmp = tmp + -1;
}
if ((boundarylen == 2) || (boundarylen == 0x46)) {
  dtrap();
  ht_senderr(hp,400,"Illegal Boundary.");
  return -1;
}
phVar6->boundarylen = boundarylen;
*tmp = '\0';
phVar6->pbuflen = boundarylen + 0x77;
                /* HEAP UNLINK HERE */
pcVar3 = (char *)npalloc(phVar6->pbuflen);
```

*Figure 24 – wbs_post() excerpt, showing the unlink operation*

Note that due to the limitations on the input buffer (only characters above 0x20 can be used), we cannot make the previous size of the overflown buffer point directly to our fake chunk (and thus our hacked FP/BP values). Therefore, the PoC relies on spraying the device's heap memory with controlled packets containing fake heap chunks (after the device is reset) and then using big negative values as the previous size to make the resulting pointer end up at one of our previously stored fake chunks. In these fake chunks, we are free to specify any FP/BP values as before, and exploitation is identical to Section 7.2.

# 9. Conclusion – Lessons Learned and the Way Ahead

### 9.1. Vulnerability discovery

After more than a year of Project Memoria, we can say that many of the vulnerabilities in embedded TCP/IP stacks are very predictable. We have distilled a large body of knowledge of existing anti-patterns in TCP/IP stack implementations, which can be used as a guide for researchers and developers to find and fix new vulnerabilities.

In AMNESIA:33, we first presented statistics about affected components and discussed initial anti-patterns. In NUMBER:JACK, we showed how a single simple issue (predictable ISN generation) tends to repeat across many stacks. In NAME:WRECK, we showed the same for more complicated patterns on DNS and started paving the way to automation by sharing static analysis queries that help developers locate potential issues in their code. Since then, more similar vulnerabilities have been found by other researchers, such as CVE-2021-26675, affecting a component used in Tesla cars.

In INFRA:HALT, we look back at all the previous anti-patterns and see most of them repeating in a single stack. We also look further into automated vulnerability discovery by working with JFrog's analysis platform.

We believe that the cybersecurity community is at a turning point, and soon automated vulnerability discovery techniques will become more common, which should make finding very large-scale vulnerabilities, such as those affecting TCP/IP stacks, faster and more frequent. All these vulnerabilities, however, will have to be disclosed, mapped to affected devices and mitigated.

### 9.2. Vulnerability disclosure

The disclosure process for INFRA:HALT took more than 9 months from initial contact with HCC Embedded on September 22, 2020, to public announcement. It took 17 days to get an initial response from HCC Embedded, weeks to convince them of the issues, as well as several months to discuss the vulnerabilities and patches (which were postponed from January to March to May 2021) and for the coordinating agencies to notify downstream device vendors.

This extended timeline – more than three times the industry-accepted 90 days – reflects the current process of coordinated vulnerability disclosure (CVD) applied to large-scale issues affecting embedded devices. Many vendors of embedded technology have software that is decades old and in different stages of support, from completely supported to end-of-life, with several levels of contracted support in between.

At the same time, there is no transparency into software components used by devices, so manually understanding if a device is affected takes a long time.

A major drawback of the current CVD process is that it misses some key elements. Although the process involves several collaborating parties (such as researchers finding the vulnerabilities, vendors patching them and CERTs coordinating the efforts), there are usually no asset owners involved, and there is no public information about the security response process of software and device vendors.

The lack of involvement of asset owners means that researchers face a dilemma: They must choose to disclose privately to a few owners they may know are affected or withhold the information until everything becomes public (assuming the vendor will want to fix the issues and has agreed to a disclosure date). Clearly, neither choice is in the best interest of the community at large.

The lack of public information about vendor security maturity means that researchers never know, when approaching a vendor, how their work will be received and how long the process will take. Mature vendors often welcome security research and are used to working with the different parties involved in CVD. Vendors that are new to the process may be hostile toward security research and have difficulties in understanding that the CVD process is helpful for a large set of stakeholders.

Both issues combined mean that there are still not enough incentives for software vendors to efficiently and effectively deal with the consequences of insecure software, which puts their customers in danger. Taking steps to bridge these gaps would put pressure on vendors and lead to more secure software.

### 9.3. Identifying vulnerable devices

After AMNESIA:33, we realized that security issues in TCP/IP stacks and applications built upon them have large implications. Hardware and software vendors often include third-party software components into their products years before vulnerabilities are found in one of these upstream components. Successful vulnerability patching depends on whether each vendor can quickly identify which of their products are affected.

There has been a plethora of industrial and academic research on tracking security issues in third-party software dependencies. However, they cannot be easily applied for embedded systems for several reasons:

- **Lack of Software Bill of Materials (SBOM) for embedded systems**. SBOMs can be used to identify the usage of vulnerable components and versions in specific devices. Software package management systems may help to construct SBOMs; however, this mainly applies to open-source software, and there is no widely adopted package manager for C (especially, in the embedded systems domain).

Surely, there was not one a year ago when many of the software components of embedded systems used nowadays were released. Not every vendor might have kept SBOM records of hardware/software released over the years, and not every vendor may agree to share this information openly.

- **Software decay and evolution**. The vulnerable upstream software component may be heavily modified over the years for a multitude of reasons. Moreover, the boundaries between different versions of TCP/IP stacks and their components may be often blurred, which may happen due to intellectual property acquisition, lack of ownership and support or the fact that downstream vendors may heavily modify the original code to meet their custom requirements. In other words, there is no easy way of comparing the vulnerable upstream code with the vendor's code in question. For closed-source systems, this option may even be unavailable.
- **Long half-life of vulnerabilities.** Many vulnerabilities are introduced into software projects at the very beginning of their existence, while embedded systems have very long lifespans. Many devices may be parts of critical infrastructure for years after they reach the end of their support lifecycle. This makes it even more challenging to identify the affected devices and issue patches for them.

We have been asked by various vendors, who suspected they could have included vulnerable TCP/IP stacks and applications into their code-bases, to help them to identify vulnerable devices in their product lines. This highlights the importance of having an effective and scalable approach to this problem.

Our goal was to find a way to identify the presence of the vulnerable upstream component – a TCP/IP stack in question. Having such a tool in their arsenal, device vendors and network operators can focus their attention on only a subset of devices rather than examining each device individually. Therefore, we have released our **open-source detector tool**[10] that allows us to identify whether a particular embedded TCP/IP stack is used in a device.

The tool is inspired by several well-known active network fingerprinting tools, such as: **nmap**[11] and **xprobe**[12]. Since the same embedded TCP/IP stacks can be a part of different real-time operating systems, traditional OS fingerprints used in **nmap** (e.g., Time-to-Live and initial TCP Window size values) may be unreliable. We observed that some fingerprints correspond to values typical to most Unix/Linux systems, and all devices running embedded TCP/IP stacks that we tested were only recognized as such systems.

*10 https://github.com/Forescout/project-memoria-detector*
*11 https://nmap.org/*
*12 The first version, which is described here: https://ofirarkin.wordpress.com/xprobe/*

Therefore, we have studied implementations of several prominent embedded stacks, including NicheStack, and came up with several key observations that allow to differentiate them:

- **ICMP quirks**. After having looked at several implementations, we realized there may be significant differences in which every stack reacts to malformed ICMP packets. For example, some of the stacks will reply to an ICMP echo packet that has an incomplete header (yet the number of absent bytes that will be tolerated differ). This essentially seems to be the most accurate identification method, provided that a stack in question exhibits any quirks.
- **TCP quirks (Urgent flag handling)**. We also have noticed that several stacks have quite different ways of handling TCP packets with the Urgent flag[13] set. We also fall back to some of more traditional approaches (e.g., looking at the sequence of TCP options), which in some cases can be quite unique, as well as use the combination of Window/TTL values (same as nmap).
- **Banner grabbing.** Embedded TCP/IP stacks are often shipped with a set of pre-built applications such as HTTP, FTP client/server, or SSH server, unlike traditional operating systems that typically have a single TCP/IP stack but may run a different application on top. NicheStack is not an exception (see Figure 1); therefore, banner grabbing can be an efficient way to identify the presence of the underlying TCP/IP stack even if a stack does not exhibit any implementation quirks and every other fingerprint fails.

We discovered that NicheStack has certain ICMP quirks, as well as several applications with specific banners (e.g., FTP server, web-server, and a Telnet server) that it may be shipped with (see Figure 1). This information can be used to fingerprint the stack with a decent degree of certainty. For example, Figure 25 shows the output of our detector tool after we run it against one of the devices in our lab (using standard and verbose modes). From the output of the verbose mode, we can see that this device matches the ICMP fingerprint, and it happens to run a webserver that has a matching banner. This combination of finger-prints allows us to conclude that this device may indeed run NicheStack (which is the TCP/IP stack that it actually runs). We are constantly working on improving the capabilities of our detector.

*13 https://datatracker.ietf.org/doc/html/rfc6093*

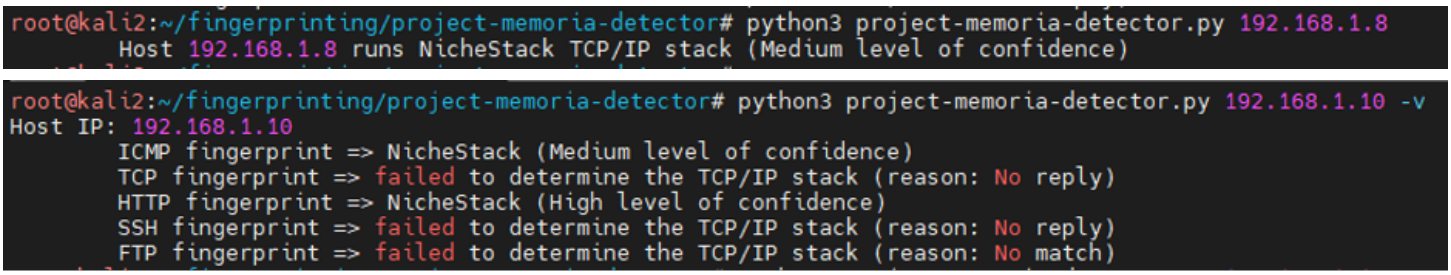



*Figure 25 – Results of the open-source fingerprinting tool against a device in the lab*

### 9.4. Vulnerability mitigation

Given the status of vulnerability discovery and disclosure, there is a need for several levels of vulnerability mitigation in the IoT/OT world, from improving the code quality of embedded software to hardening network configurations.

Both software and device vendors need to adopt secure software development lifecycles and improve their security response processes. As we mentioned in Section 9.1, automated vulnerability discovery is not only for security researchers but mainly for software developers to eliminate bugs and to improve the products they ship.

Also, as mentioned in Section 7, we are still not seeing basic mitigation techniques being applied in the IoT/OT world, such as: stack and heap canaries, address space randomization, no-execute memory pages, and format string attack mitigations.

In the IT world (PC, server and mobile ecosystems), these techniques have been the standard for decades. It is unacceptable to keep ignoring them when modern processing power enables using them fully without sacrificing any functionality.

The reality is that this will take a long time, which means there will still be insecure devices in critical networks in the foreseeable future. Asset owners must ensure that these devices are not easily accessible by attackers and that network traffic is closely monitored to detect problems at the earliest stages.