

How TCP/IP Stacks Breed Critical Vulnerabilities in IoT, OT and IT Devices



Written by Daniel dos Santos, Stanislav Dashevskiy, Jos Wetzels and Amine Amri



A note on vulnerability disclosure

We would like to thank the [CERT Coordination Center](#), the [ICS-CERT](#), the German [Federal Office for Information Security \(BSI\)](#) and the [JPCERT Coordination Center](#) for their help in coordinating the disclosure of the AMNESIA:33 vulnerabilities.

We do not provide a list of affected or suspected-to-be-affected vendors in this report. We have shared this list with the coordinating agencies, and we will rely on them, as well as the vendors, to provide their own advisories.

We do mention a few examples of vendors, including components and devices that embed the vulnerable stacks, in Chapter 5. Those are mentioned because they help to illustrate important points.

1. Executive summary

- [ForeScout Research Labs](#) has launched [Project Memoria](#), an initiative that aims at providing the community with the **largest study on the security of TCP/IP stacks**. Project Memoria's goal is to develop the understanding of common bugs behind the vulnerabilities in TCP/IP stacks, identifying the threats they pose to the extended enterprise and how to mitigate those.
- **AMNESIA:33** is the first study we have published under Project Memoria. In this study, we discuss the results of the security analysis of seven **open source TCP/IP stacks** and report a bundle of **33 new vulnerabilities** found in four of the seven analyzed stacks that are used by major IoT, OT and IT device vendors.
- **Four of the vulnerabilities in AMNESIA:33 are critical**, with potential for remote code execution on certain devices. Exploiting these vulnerabilities could allow an attacker to take control of a device, thus using it as an entry point on a network for internet-connected devices, as a pivot point for lateral movement, as a persistence point on the target network or as the final target of an attack. For enterprise organizations, this means they are at increased risk of having their network compromised or having malicious actors undermine their business continuity. For consumers, this means that their IoT devices may be used as part of large attack campaigns, such as botnets, without them being aware.
- AMNESIA:33 affects **multiple open source TCP/IP stacks** that are **not owned by a single company**. This means that a single vulnerability tends to **spread easily and silently** across multiple codebases, development teams, companies and products, which presents significant challenges to patch management.
- We estimate that more than 150 vendors and millions of devices are vulnerable to AMNESIA:33. However, **it is difficult to assess the full impact** of AMNESIA:33 because the vulnerable stacks are widely spread (across different IoT, OT and IT devices in different verticals), highly modular (with components, features and settings being present in various combinations and code bases often being forked) and incorporated in undocumented, deeply embedded subsystems. For the same reasons, these vulnerabilities tend to be very hard to eradicate.
- The TCP/IP stacks affected by AMNESIA:33 can be found in operating systems for embedded devices, systems-on-a-chip, networking equipment, OT devices and a myriad of enterprise and consumer IoT devices.
- TCP/IP stacks are critical components of all IP-connected devices, including IoT and OT, since they enable basic network communications. A security flaw in a TCP/IP stack can be extremely dangerous because the code in these components may be used **to process every incoming network packet that reaches a device**. This means that some vulnerabilities in a TCP/IP stack allow for a device to be exploited, even when it simply sits on a network without running a specific application.
- Many of the vulnerabilities reported within **AMNESIA:33** arise from bad software development practices, such as an absence of basic input validation. They relate mostly to **memory corruption** and can cause **denial of service, information leaks** or **remote code execution**.
- Due to the complexity of identifying and patching vulnerable devices, vulnerability management for TCP/IP stacks is becoming a challenge for the security community. We recommend **adopting solutions that provide granular device visibility**, allow the monitoring of network communications and isolate vulnerable devices or network segments to manage the risk posed by these vulnerabilities.

2. About Project Memoria

Forescout Research Labs worked in close collaboration with JSOF to [identify vendors and devices potentially affected by the Ripple20 vulnerabilities](#), which affect the Treck TCP/IP stack and many vendors that use it in their IoT/OT products.

Ripple20 is the latest example of TCP/IP stack vulnerabilities that expose a complex IoT supply chain, thus affecting millions of devices across many industries. While working on Ripple20, it became clear that the problems with TCP/IP security flaws are not related to a few vendor-specific stacks. On the contrary, we hypothesized that the problem was much more generic and widespread.

Forescout Research Labs has launched Project Memoria, an initiative with the mission of providing the community with the largest study on the security of TCP/IP stacks. Under Project Memoria, Forescout Research Labs collaborates with industry peers, as well as universities and research institutes, to understand common mistakes

behind the vulnerabilities in TCP/IP stacks, identify the threats they pose to the extended enterprise and how to mitigate the risk.

This report focuses on AMNESIA:33, the first study we published under Project Memoria, where we discuss the results of the security analysis of seven open source TCP/IP stacks. Other studies, focusing on different TCP/IP stack components, will be ongoing.

With AMNESIA:33, we report on 33 new vulnerabilities found in four of the seven stacks analyzed, namely [uIP](#), [FNET](#), [PicoTCP](#) and [Nut/Net](#). These stacks exist in several variants, and they are used by several vendors in different commercial products, both consumer and enterprise grade, including critical devices in OT environments.

The vulnerabilities included in AMNESIA:33 range in potential impact from denial of service to remote code execution and affect several components and features of the stacks, such as IPv4 and v6, ICMP, TCP and DNS.

INFORMATIONAL

The origins of Project Memoria

The word **Memoria** originates from Latin and, in many Romance languages, means “memory.” Its use as a project name refers to two facts. First, that many vulnerabilities in TCP/IP stacks bring to memory vulnerabilities we used to see in IT systems in the 1990s and early-2000s. Second, that TCP/IP stacks are a somewhat forgotten foundation of the IoT that we want to bring forward from memory.

The word **Amnesia** refers to the fact that most vulnerabilities in TCP/IP stacks, particularly the ones in AMNESIA:33, stem from memory corruption, which is an attacker’s capability of reading or writing memory locations that were not intended in the original behavior of a target software. The different degrees of control over that capability are what lead to different impacts such as denial of service, information leaks and remote code execution.

INFORMATIONAL

**A look at the foundations:
the importance of TCP/IP stacks**

Traditionally, embedded systems – such as a combination of hardware and software designed for a specific function, such as sensors or control mechanisms – relied on serial networks for communication. With the rise of the Internet of Things and the convergence of OT and IT, networks based on the TCP/IP protocol suite have proliferated, and they have often displaced old serial networks.

Nowadays, devices communicate with each other via a wide array of protocols at different layers. The TCP/IP model in Figure 1 shows how network communications happen in layers and how each layer relies on the others. You can see how application layer communications (such as HTTP) rely on underlying internet layer (IP) and transport layer (TCP) communications. The TCP/IP layers are of paramount importance in modern network communication because they are at the very foundation of every communication happening via any of the protocols above.

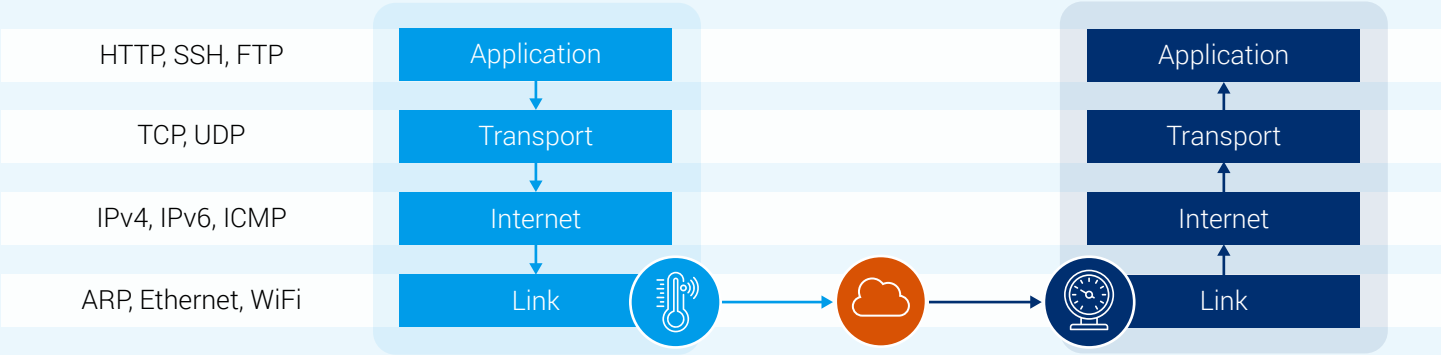


Figure 1 – The TCP/IP networking model

TCP/IP functionality is implemented by means of a piece of software called a protocol stack. Protocol stacks, whether general purpose [like SMB](#) or domain specific like [DNP3](#), present an attractive target for malicious actors because (a) they have direct network exposure; (b) they are often implemented as low-level system functionality and, as such, tend to be implemented in [memory-unsafe languages such as C and C++](#); (c) they are widely deployed; and (d) they often offer a variety of unauthenticated functionality exposing potential attack surface.

In addition, the code at the lower layers of protocol stacks (like TCP/IP) is used to process every incoming frame and

packet that reaches a device, allowing for cases where a system can be exploited even when it is not running a specific application or listening on a particular port.

Finally, embedded systems, such as IoT and OT devices, tend to have long vulnerability lifespans resulting from a combination of patching issues, long support lifecycles and vulnerabilities ‘trickling down’ highly complex and opaque supply chains. As a result, vulnerabilities in embedded TCP/IP stacks have the potential to affect millions – even billions – of devices across verticals and tend to remain a problem for a very long time.

3. AMNESIA:33 – a security analysis of open source TCP/IP stacks

3.1. Why focus on open source TCP/IP stacks?

If “software is eating the world,” as Marc Andreessen [famously said](#), then we can say that open source software is eating the embedded/IoT world. The [2019 Embedded Markets Study](#) revealed that **88% of embedded projects reused source code** (either internally developed code, third-party open source code or third-party commercial code). This is not so surprising, since development projects rarely start from scratch. What is more interesting is that, **out of these projects, 58% used an open source RTOS** (16% with and 42% without commercial support), **which typically includes an open source embedded TCP/IP stack**. Even more interesting, a historical analysis in the same study shows that the use of commercial OSes is declining since at least 2015, while the use of open source is increasing. Sixty-three percent of respondents in that study claimed that they intend to use an open source OS in their next project.

One reason why open source components are so popular is that [35% of developers](#) consider the availability of full source code to be the most important reason to choose an OS. Regardless of the reasons, the same report shows that only 4% of design time is spent on security and privacy assessments.

There is a wealth of literature pointing at the risks posed by third-party components on enterprise software, [including open source](#), such as the need to keep track of vulnerabilities in these components, assess their impact in a final product and decide whether or not to fix them, assuming that a fix is provided by the third party. With AMNESIA:33, we aim to show how widespread these issues are among open source TCP/IP stacks.

3.2. Which open source stacks, exactly?

For our study, we selected a sample of seven open source embedded TCP/IP stacks to analyze. Our choices were based on (i) whether the stack is used or supported by popular RTOSes (e.g., [FreeRTOS](#) and [uC/OS](#) are used in respectively [18% and 7% of embedded projects](#)); and (ii) the popularity of embedded devices using the stack.

Table 1 lists the stacks that we ultimately selected for our analysis. Note that (1) most of the stacks are close to two decades old, which means that many versions of their code exist, and many devices using the stacks are probably end-of-life; and (2) we only provide examples of notable OSes using the stacks, so the list is far from being exhaustive.

Table 1 – Analyzed stacks

#	Stack	Description	License	Examples of OSes using it
1	lwIP	lwIP (Lightweight IP) was developed in 2000 by Adam Dunkels at the Swedish Institute of Computer Science and is now maintained by a large group of developers. lwIP has become very popular as part of FreeRTOS or as a standalone stack, now being used by products such as Tesla gateway ECU and Philips Hue smart lights.	BSD	LiteOS RT-Thread FreeRTOS ReactOS
2	uIP	uIP (micro IP) was designed to be even lighter than lwIP and was released in 2001 as an open source project also by Adam Dunkels. It was extended by Cisco in 2008 with IPv6. Its development has been halted as a standalone project, but it continues as part of the Contiki OS, which in turn has a new version called Contiki-NG. uIP is known to have been used in devices as diverse as networking switches and picosatellites.	BSD	Contiki Contiki-NG RT-Thread FreeRTOS NuttX
3	Nut/Net	Nut/Net is the IP stack used by NutOS, which has been developed by the Ethernut project since 2002. The stack is used both by hobbyists and by commercial devices , including in QT/ICS .	BSD	NutOS
4	FNET	FNET was developed originally at Freescale in 2003 and made public in 2009. It is currently maintained by Andrey Butok.	Apache v2	-
5	picoTCP	picoTCP was developed by Altran Intelligent Systems and made open source in 2013. The stack continues to be developed as picoTCP-NG, which is no longer supported by Altran. Online, there is not much public mention of PicoTCP uses.	GPLv2 / Commercial	seL4 TRENTOS
6	CycloneTCP	CycloneTCP is developed by Oryx Embedded and distributed in source code form since 2013. Its website mentions uses in Industrial IoT, energy metering and management, transportation and smart buildings.	GPLv2 / Commercial	-
7	uC/TCP-IP	uC/TCP-IP was developed originally by Micrium in 2002 and has been open sourced in February 2020. uC/OS, which typically relies on the stack, is known to be used in medical devices, industrial control systems and other critical applications.	Apache v2 (previously commercial)	uC/OS-II uC/OS-III Cesium

3.3. 33 new findings

To perform our analysis, we used a combination of automated fuzzing (white-box code instrumentation based on [libFuzzer](#)), manual analysis guided by variant hunting using the [Joern](#) code querying engine and a pre-existing corpus of vulnerabilities that we will describe in Chapter 4 and manual code review. In this way, we

found a total of 33 new vulnerabilities spread across the different stacks, as described in Table 2. In the Table, we also differentiate between vulnerabilities found via fuzzing (33%) and vulnerabilities found via static analysis (67%). Although the targets are different, these numbers are in line with the results reported by [Google's 'Project Zero' at Black Hat USA 2019](#).

Table 2 – AMNESIA:33 vulnerabilities across the analyzed stacks

Stack	Versions Analyzed	# of Total New Vulnerabilities	# of New Vulnerabilities Found Via Fuzzing	# of New Vulnerabilities Found Via Code Analysis
uIP	uIP 1.0, Contiki 3.0, Contiki-NG 4.5	13	6	7
picoTCP	picoTCP 1.7.0, picoTCP-NG 2.0.0	10	5	5
FNET	4.6.3	5	0	5
Nut/Net	5.1	5	0	5
lwIP	2.1.2	0	0	0
CycloneTCP	1.9.6	0	0	0
uC/TCP-IP	3.06.00	0	0	0

HIGHLIGHTS

In our study, we did not find any vulnerability in the lwIP, CycloneTCP and uC/TCP-IP stacks. Although this does not imply that there are no flaws in these stacks, we observed that the three stacks have very consistent bounds checking and generally do not rely on shotgun parsing, one of the most common anti-patterns we identified.

We are not aware of any previous research done or vulnerabilities found on CycloneTCP or uC/TCP-IP. lwIP has a previously reported DNS cache poisoning ([CVE-2014-](#)

[4883](#)) and has been [analyzed before](#) as part of FreeRTOS and [in terms of TCP conformance](#). We found two bugs in the PPP component of lwIP, but they were reported to the project and found unexploitable. As usual, these negative results do not imply that there are no vulnerabilities in these three stacks, but, from our analysis, they seem generally more robust than the others in our dataset.

The details of the new vulnerabilities are shown in Table 3 (uIP), Table 4 (picoTCP), Table 5 (FNET) and Table 6 (Nut/Net), and can be summarized as follows:

- AMNESIA:33 affects seven different components of the stacks (DNS, IPv6, IPv4, TCP, ICMP, LLMNR and mDNS). Two vulnerabilities in AMNESIA:33 only affect 6LoWPAN wireless devices.
- AMNESIA:33 has four categories of potential impact: remote code execution (RCE), denial of service (DoS via crash or infinite loop), information leak (infoleak) and DNS cache poisoning. Generally, these vulnerabilities can be exploited to take full control of a target device (RCE), impair its functionality (DoS), obtain potentially sensitive information (infoleak) or inject malicious DNS records to point a device to an attacker-controlled domain (DNS cache poisoning).

Table 3 – Details of the new vulnerabilities on uIP

● Low: 0.1-3.9, ● Medium: 4.0-6.9 ● High: 7.0-8.9 ● Critical: 9.0-10.0

CVE-2020-	Description	Affected Component	Potential Impact	CVSSv3.1 Score
13984	The function used to process IPv6 extension headers and extension header options can be put into an infinite loop state due to unchecked header/option lengths.	Ext. header parsing in IPv6 (6LoWPAN)	DoS	7.5
13985	The function used to decapsulate RPL extension headers does not check for unsafe integer conversion when parsing the values provided in a header, allowing attackers to corrupt memory.	Ext. header parsing in IPv6	DoS	7.5
13986	The function used to decapsulate RPL extension headers does not check the length value of an RPL extension header received, allowing attackers to put it into an infinite loop.	Ext. header parsing in IPv6 (6LoWPAN)	DoS	7.5
13987	The function that parses incoming transport layer packets (TCP/UDP) does not check the length fields of packet headers against the data available in the packets. Given arbitrary lengths, an out-of-bounds memory read may be performed during the checksum computation.	TCP/UDP checksum calculation in IPv4	DoS Infoleak	8.2
13988	The function that parses the TCP MSS option does not check the validity of the length field of this option, allowing attackers to put it into an infinite loop, when arbitrary TCP MSS values are supplied.	TCP options parsing in IPv4	DoS	7.5
17437	When handling TCP Urgent data, there are no sanity checks for the value of the Urgent data pointer, allowing attackers to corrupt memory by supplying arbitrary Urgent data pointer offsets within TCP packets.	TCP packet processing	DoS	8.2

CVE-2020-	Description	Affected Component	Potential Impact	CVSSv3.1 Score
17438	The code that reassembles fragmented packets does not validate the total length of an incoming packet specified in its IP header, as well as the fragmentation offset value specified in the IP header. This may lead to memory corruption.	Fragmented packet reassembly in IPv4	DoS	7.0
17439	Incoming DNS replies are parsed by the DNS client even if there were no outgoing queries. The DNS transaction ID is not sufficiently random. Provided that the DNS cache is quite small (4 entries), this facilitates DNS cache poisoning attacks.	DNS response processing	DNS cache poisoning	8.1
17440	When parsing incoming DNS packets, there are no checks whether domain names are null-terminated. This allows attackers to achieve memory corruption with crafted DNS responses.	DNS domain name decoding	DoS	7.5
24334	The code that processes DNS responses does not check whether the number of responses specified in the DNS packet header correspond to the response data available in the DNS packet, allowing attackers to corrupt memory.	DNS response processing	DoS	8.2
24335	The function that parses domain names lacks bounds checks, allowing attackers to corrupt memory with crafted DNS packets.	DNS domain name decoding	DoS	7.5
24336	The code for parsing DNS records in DNS response packets sent over NAT64 does not validate the length field of the response records, allowing attackers to corrupt memory.	DNS response parsing in NAT64	RCE	9.8
25112	Several issues, such as insufficient checks for the IPv4/IPv6 header length and inconsistent checks for the IPv6 header extension lengths, allow attackers to corrupt memory.	ICMPv6 echo/reply processing	RCE	8.1

Table 4 – Details of the new vulnerabilities on picoTCP

CVE-2020-	Description	Affected Component	Potential Impact	CVSSv3.1 Score
17441	The payload length field of IPv6 extension headers is not checked against the data available in incoming packets, allowing attackers to corrupt memory.	Ext. header parsing in IPv6, ICMPv6 checksum	DoS Infoleak	7.5
17442	The function that processes the Hop-by-Hop extension header in IPv6 packets and its options lacks any checks against the length field of the header, allowing attackers to put the function into an infinite loop by supplying arbitrary length values.	Ext. header parsing in IPv6	DoS	7.5
17443	When processing ICMPv6 echo requests, there are no checks for whether the ICMPv6 header consists of at least 8 bytes (set by RFC443). This leads to the function that creates ICMPv6 echo replies based on a received request with a smaller header to corrupt memory.	ICMPv6 echo request processing	DoS	8.2
17444	The function that processes IPv6 headers does not check the lengths of extension header options, allowing attackers to put this function into an infinite loop with crafted length values.	Ext. header parsing in IPv6	DoS	7.5
17445	The function that processes the IPv6 Destination Options extension header does not check the validity of its options lengths, allowing attackers to corrupt memory and/or put the function into an infinite loop with crafted length values.	Ext. header parsing in IPv6	DoS	7.5
24337	The function that processes TCP options does not validate their lengths, allowing attackers to put the function into an infinite loop with uncommon/unsupported TCP options that have crafted length values.	TCP options parsing in IPv4	DoS	7.5
24338	The function that parses domain names lacks bounds checks, allowing attackers to corrupt memory with crafted DNS packets.	DNS domain name decoding	RCE	9.8
24339	The function that parses domain names lacks bounds checks, allowing attackers to corrupt memory with crafted DNS packets.	DNS domain name decoding	DoS	7.5

CVE-2020-	Description	Affected Component	Potential Impact	CVSSv3.1 Score
24340	The code that processes DNS responses does not check whether the number of responses specified in the DNS packet header correspond to the response data available in the DNS packet, allowing attackers to perform memory corruption.	DNS response processing	DoS Infoleak	8.2
24341	The TCP input data processing function does not validate the length of incoming TCP packets, allowing attackers to read out of bounds and perform memory corruption.	TCP packet processing	DoS Infoleak	8.2

Table 5 – Details of the new vulnerabilities on FNET

CVE-2020-	Description	Affected Component	Potential Impact	CVSSv3.1 Score
17467	When parsing LLMNR requests, there are no checks whether domain names are null-terminated. This may allow attackers to read out of bounds.	LLMNR state machine	Infoleak	8.2
17468	The function that processes the IPv6 Hop-by-Hop extension header does not check the validity of its options lengths, allowing attackers to corrupt memory.	Ext. header parsing in IPv6	DoS	7.5
17469	The IPv6 packet reassembly function does not check whether the received fragments are properly aligned in memory, allowing attackers to perform memory corruption with crafted IPv6 fragmented packets.	Fragmented packet reassembly in IPv6	DoS	5.9
17470	The code that initializes the DNS client interface structure does not set sufficiently random transaction IDs (they will be always set to 1), facilitating DNS cache poisoning attacks.	DNS response processing	DNS cache poisoning	4
24383	When parsing incoming mDNS packets, there are no checks whether domain names are null-terminated. This allows attackers to achieve memory corruption and/or memory leak.	DNS domain name decoding	DoS Infoleak	6.5

Table 6 – Details of the new vulnerabilities on Nut/Net

CVE-2020-	Description	Affected Component	Potential Impact	CVSSv3.1 Score
25107	The code that processes DNS questions/responses has several issues: (1) there is no check on whether a domain name is NULL-terminated; (2) the DNS response data length is not checked (can be set to arbitrary value from a packet); (3) the number of DNS queries/responses (set in DNS header) is not checked against the data present; (4) the length byte of a domain name in a DNS query/response is not checked and is used for internal memory operations.	DNS domain name decoding/ DNS response processing	DoS	7.5
25108			DoS	7.5
25109			DoS	8.2
25110			DoS	8.2
25111			RCE	9.8

INFORMATIONAL

A note on bug collision

All the vulnerabilities described in this chapter were found independently. However, [CVE-2020-24338](#) was later found to have been reported previously as CVE-2017-1000210, which was fixed on PicoTCP-NG and had a [pull request](#)

on the original project fixing it that was never merged on the master branch. Similarly, CVE-2020-17437 was later found to have been [fixed on Contiki-NG](#) but not on previous versions (uIP and Contiki) and never reported as a CVE.

4. A comparison with similar studies

General-purpose TCP/IP stacks seem to have become more robust since the days of [WinNuke](#) and the [Ping of Death](#) affecting Linux, Mac and Windows systems, despite [occasional issues still occurring](#).

However, in the past few years, together with AMNESIA:33, there has been a spate of vulnerabilities in various embedded TCP/IP stacks as shown in Table 7. Throughout this chapter, we will use the sample of vulnerabilities shown in the table to perform an analysis of the general trend.

Table 7 – Vulnerabilities in embedded TCP/IP stacks over the years

Year of disclosure	TCP/IP stack	# Vulnerabilities disclosed
2013	Microchip TCP/IP	1
2014	uIP, lwIP	1
2017	RTCS TCP/IP	2
2017	picoTCP	1
2018	FreeRTOS+TCP	10
2019	Nucleus NET	1
2019	Interpeak IPnet	11 (URGENT/11)
2020	InterNiche NicheStack	1
2020	Treck TCP/IP	19 (Ripple20)
2020	uIP, PicoTCP, FNET, Nut/Net	33 (AMNESIA:33)

The first thing to notice from Table 7 is the significant increase in numbers of vulnerabilities in recent advisories, particularly for something as fundamental as a TCP/IP stack. This indicates relative security immaturity, which does not correlate with adoption, since [IPnet](#) and [Treck](#), for instance, are highly popular stacks that have been in active use for decades. As we will show in Chapter 5, the same is true for the stacks of AMNESIA:33.

To better understand and contextualize the AMNESIA:33 vulnerabilities using the dataset of Table 7, we analyze those vulnerabilities from five angles:

- **Affected components**, i.e., which parts of stacks are usually vulnerable. We find that the DNS, TCP and IP sub-stacks are the most often vulnerable. DNS, in particular, seems to be vulnerable because of its complexity.
- **Types of vulnerabilities**, i.e., which vulnerabilities are often found in these stacks. The most common memory corruption vulnerabilities are out-of-bounds

reads and writes, followed by integer overflows. The most common non-memory-related issue is state confusion arising from request-reply matching.

- **Anti-patterns**, i.e., what code patterns are most conducive to vulnerabilities. We find that common anti-patterns include issues with calculating and/or validating header and field lengths, properly parsing various header option fields, ensuring that there is enough data in the packet (in contrast to relying on what is specified in the header), handling the TCP Urgent pointer, fragmentation reassembly, IP tunneling, verification of DNS domain name length and null termination.
- **Exploitability**, i.e., if and how these vulnerabilities can be exploited. In general, exploitability comes down to how a stack is used in a particular device, which can be broken down in three major categories: stack configuration (such as which components are used and how they are used), networking hardware

- and driver (such as what functions are offloaded to dedicated hardware) and the target platform (such as CPU architecture).
- **Potential impact**, i.e., what the impact can be of exploiting these vulnerabilities. We discuss that although most vulnerabilities in TCP/IP stacks are denials of service (that might be seen as non-critical), impact is highly contextual to a specific device and use case (e.g., a DoS is very dangerous in mission-critical devices).

4.1. Which components are typically flawed?

A typical TCP/IP stack is composed of different parts that handle different protocols, which we hereby call *components*. In Figure 2, we can see that the **most affected components in our sample of vulnerabilities are the DNS, TCP and IPv4/IPv6 sub-stacks**, followed by DHCP, ICMP/ICMPv6, ARP and others. The only vulnerability that stands out is [CVE-2020-11904](#) (part of Ripple20), which was discovered within the memory allocator component used by the Treck stack. Most of the vulnerabilities in AMNESIA:33 impact the DNS, IPv6 and TCP components.

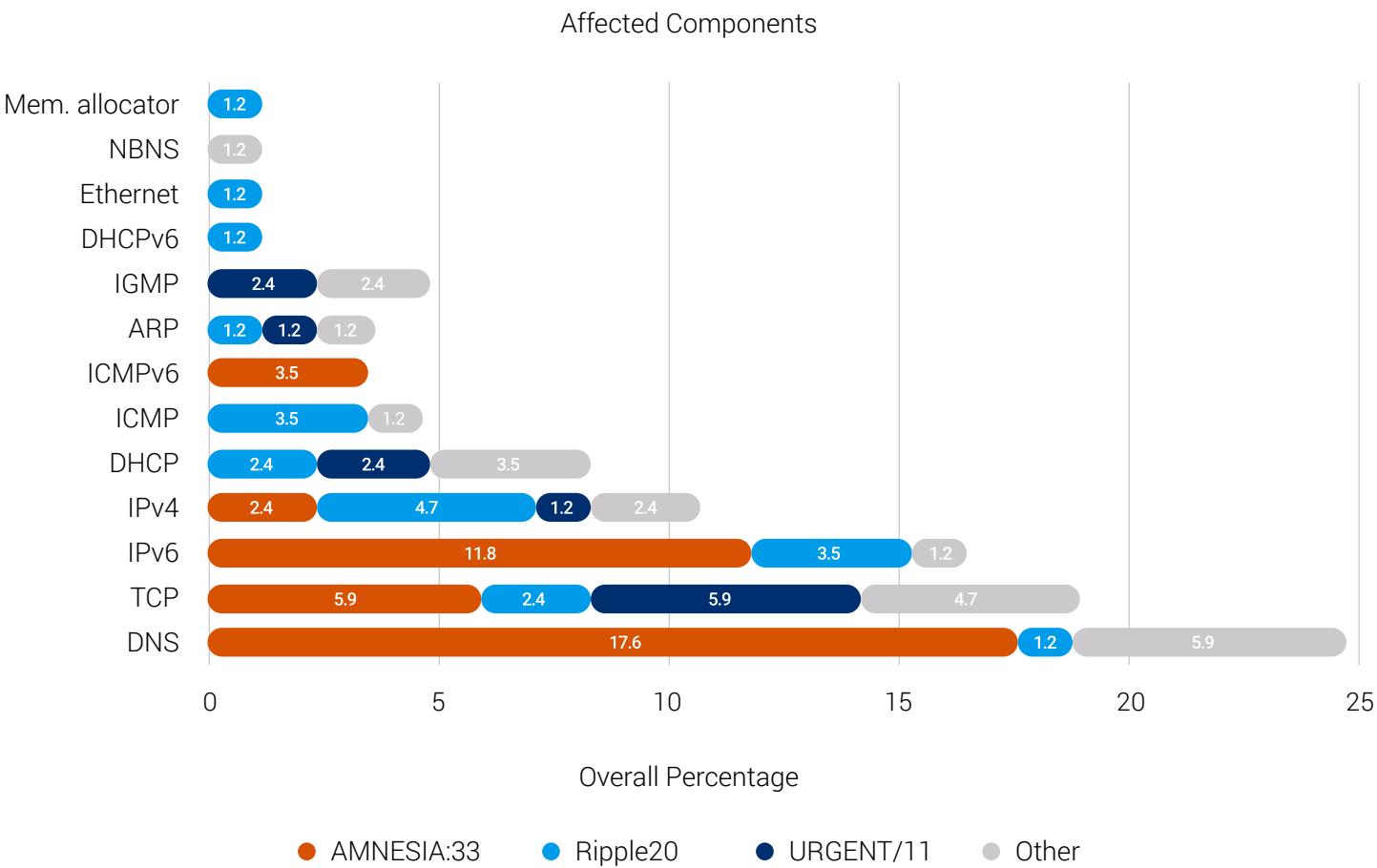


Figure 2 – Components of popular TCP/IP stacks affected by vulnerabilities

Vulnerabilities in the IP and TCP substacks are particularly interesting since they are independent of applications running on top of them. Vulnerabilities in the IP substack do not even require a TCP or UDP port to be open for a device to be exploited. Some vulnerable implementations also first attempt to fully parse incoming TCP/UDP packets before checking for existing connections, allowing them to be exploited even when there are no open ports.

DNS appears to be a vulnerability-prone component because it is a complex, feature-rich protocol, different from many other components in the stack. Indeed, the DNS component is a client that usually communicates with a few standard servers rather than a server that communicates with many other clients; this may lead to errors in the implementations. A possible mitigation of this complexity that we have seen is the OpenBSD

implementation, which relies on [pledges](#) to isolate DNS processing and limit the effects of vulnerabilities. However, as we will discuss below, this kind of mitigation is rare in embedded devices.

4.2. What are the most common vulnerability types?

Several [taxonomies of vulnerability types](#) exist in the literature, which can be more or less granular.¹ To simplify our analysis, we use a categorization containing the five most common memory-related vulnerability types we observed, one common type that is not related to memory (state confusion) and a category of “other” less common issues that are not related to memory corruption (such as race conditions and improper random number generation).

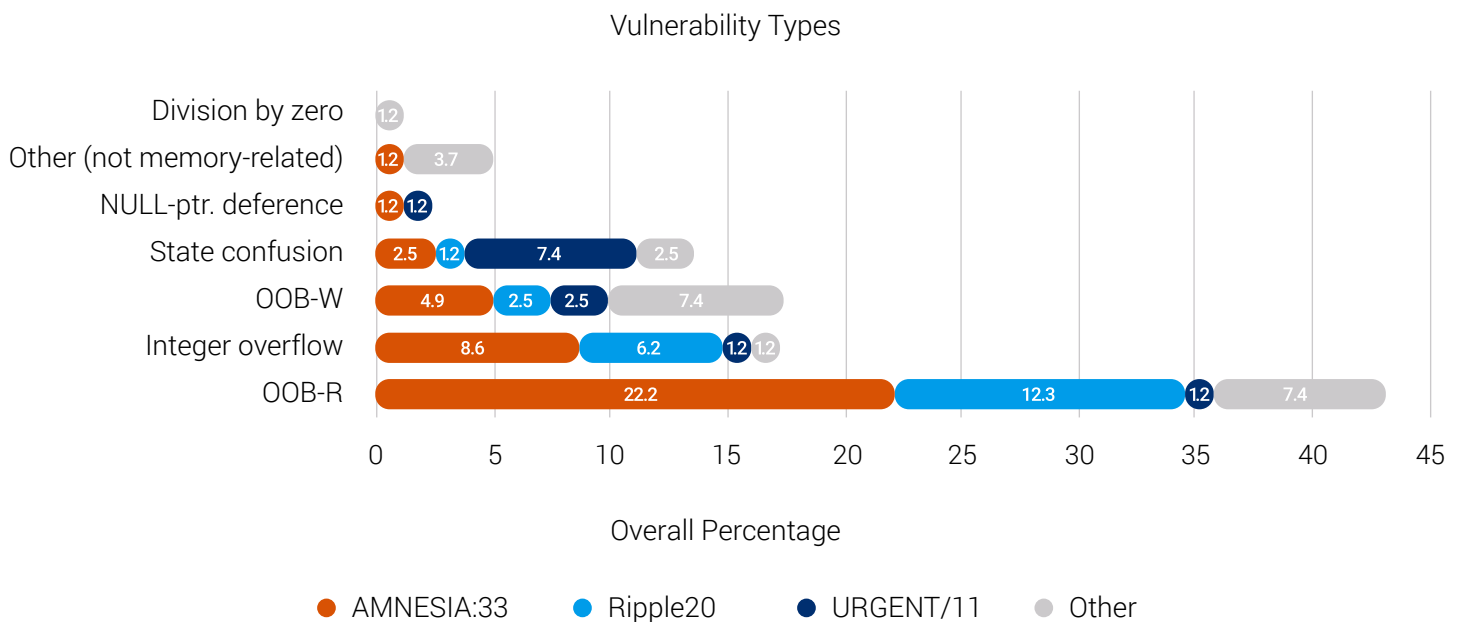


Figure 3 – Types of vulnerabilities found in popular TCP/IP stacks

¹ For example, MITRE lists almost 900 weaknesses divided into more than 300 categories in its [CWE framework](#).

Figure 3 shows a breakdown of the issues in our sample by these vulnerability types, the most important being:

- **Out-of-Bounds Read & Write (OOB-R and OOB-W):** TCP/IP and the various protocols built on top of it have a variety of attacker-controlled length and offset fields that influence memory manipulation operations and, as such, require proper memory bounds checking.
- **Integer Overflow:** The length and offset fields are often incorporated in arithmetic operations during assignments or comparisons. Given that integer representations have a fixed size, it is important to take the upper and lower bounds of that size into account before applying those operations because if a value grows larger than the maximum or smaller than the minimum, it wraps around the bound and ‘comes out the other side.’ Truncation and signedness issues because of type casting also play a role here. Often, an integer wraparound can be used as leverage to cause an out-of-bounds read or write. In some cases, integer overflows may lead to infinite loops (e.g., when parsing TCP options or IPv6 extension headers) in cases when the overflowed variable has impact on the exit condition of the loop.
- **State Confusion:** Stateful protocol handling requires carefully keeping track of the current state and session information of a given connection. Even in stateless protocols, one must typically perform request-reply matching to ensure that an incoming reply is the response to a previously sent request. Incorrectly implemented or overly permissive state machines and ambiguous protocol specifications can result in state confusion bugs, which in turn can result in a misalignment between internally stored data and expectations around subsequent incoming data.
- **Null Pointer Dereferences:** A common case of illegal or dangling pointer dereferences where an unmapped or protected memory address is read from or written

to. Depending on target memory organization, protection and fault handling this can result in Denial-of-Service or, if the dangling pointer’s contents are attacker-controllable, sometimes even Remote Code Execution.

- **Division by Zero:** This happens when an arithmetic operation has a divisor equal to zero, which is undefined behavior and can result in an error or exception.

A good example of request-reply matching issues (state confusion) in AMNESIA:33 is CVE-2020-17439, which affects uIP. The stack does not sufficiently check whether incoming DNS reply packets match the outgoing DNS queries. Attackers only need to wait until there are any outgoing DNS queries from a vulnerable device and send any DNS reply with a matching DNS transaction ID. Since transaction IDs in this stack are not properly randomized (can only be between 0x00 and 0x03), this reply will be accepted by the stack. It is trivial for attackers to leverage this vulnerability to either perform DNS cache poisoning attacks (thus injecting malicious DNS records) or to exploit other vulnerabilities that may be present within the DNS reply processing functionality (e.g., see CVE-2020-17440 and CVE-2020-24334 described below).

In the following “Technical Dive In,” we analyze at length an example in AMNESIA:33 that showcases the integer overflow and OOB-R/W types of vulnerabilities.

TECHNICAL DIVE IN

Integer Underflow and OOB-R/W in AMNESIA:33

An illustrative example of a combined integer overflow and OOB-R/W issue is CVE-2020-17443 in PicoTCP. The

vulnerable code is located within the `pico_icmp6_send_echoreply()` function that prepares ICMPv6 echo reply packets to be sent in response to incoming ICMPv6 echo requests (see Figure 4).

```

61 static int pico_icmp6_send_echoreply(struct pico_frame *echo)
62 {
63     struct pico_frame *reply = NULL;
64     struct pico_icmp6_hdr *ehdr = NULL, *rhdr = NULL;
65     struct pico_ip6 src;
66     struct pico_ip6 dst;
67
68     reply = pico_proto_ipv6.alloc(&pico_proto_ipv6, echo->dev, (uint16_t)(echo->transport_len));
69     if (!reply) {
70         pico_err = PICO_ERR_ENOMEM;
71         return -1;
72     }
73
74     echo->payload = echo->transport_hdr + PICO_ICMP6HDR_ECHO_REQUEST_SIZE;
75     reply->payload = reply->transport_hdr + PICO_ICMP6HDR_ECHO_REQUEST_SIZE;
76     reply->payload_len = echo->transport_len;
77
78     ehdr = (struct pico_icmp6_hdr *)echo->transport_hdr;
79     rhdr = (struct pico_icmp6_hdr *)reply->transport_hdr;
80     rhdr->type = PICO_ICMP6_ECHO_REPLY;
81     rhdr->code = 0;
82     rhdr->msg.info.echo_reply.id = ehdr->msg.info.echo_reply.id;
83     rhdr->msg.info.echo_reply.seq = ehdr->msg.info.echo_request.seq;
84     memcpy(reply->payload, echo->payload, (uint32_t)(echo->transport_len - PICO_ICMP6HDR_ECHO_REQUEST_SIZE));
85     rhdr->crc = 0;
86     rhdr->crc = short_be(pico_icmp6_checksum(reply));
87     /* Get destination and source swapped */
88     memcpy(dst.addr, ((struct pico_ipv6_hdr *)echo->net_hdr)->src.addr, PICO_SIZE_IP6);
89     memcpy(src.addr, ((struct pico_ipv6_hdr *)echo->net_hdr)->dst.addr, PICO_SIZE_IP6);
90     pico_ipv6_frame_push(reply, &src, &dst, PICO_PROTO_ICMP6, 0);
91     return 0;
92 }

```

Figure 4 - CVE-2020-17443

Here, the memory for an ICMPv6 response header and payload (**reply**) will be allocated based on the size of a request header and payload **echo->transport_len** (line 68); various fields of the reply packet will be set based on the echo packet (lines 74-89), and the reply will be queued for sending (line 90). In particular, the reply payload is being copied directly from the echo payload using the `memcpy()` function call (line 84). Here, the size of the memory copy is

the ICMPv6 length minus the minimum possible ICMPv6 header length of 8 bytes (defined in the **PICO_ICMP6HDR_ECHO_REQUEST_SIZE** constant).

The code that accepts ICMPv6 echo request packets (omitted for brevity) will process any packets that appear to have an ICMPv6 header, even when it is shorter than 8 bytes. In fact, it only checks that the first byte after the IP

TECHNICAL DIVE IN

header is 0x80 (ICMPv6 echo request). The attackers have explicit control over **echo->transport_len**, and if this value is shorter than 8 bytes, the arithmetic operation in the third argument of the `memcpy()` call (line 84) will underflow, resulting in a large unsigned value. `memcpy()` will write out-of-bounds of the **reply->payload**.

While [RFC 1256](#) hints that the minimum size of the ICMP message should be 8 bytes (for a router solicitation message), it is not stated explicitly that packets with ICMP

payloads smaller than 8 bytes must be discarded when processing ICMP echo request messages.

Sometimes, integer overflow bugs may not necessarily lead to OOB-R/W vulnerabilities but exist side-by-side with them, allowing attackers to achieve different goals. An illustrative example from AMNESIA:33 is CVE-2020-17437, affecting uIP (shown in Figure 5). This vulnerability stems from the misuse of the Urgent pointer, yet it is different from the vulnerabilities reported as part of URGENT/11.

```

1640  /* Check the URG flag. If this is set, the segment carries urgent
1641  data that we must pass to the application. */
1642  if((BUF->flags & TCP_URG) != 0) {
1643  #if UIP_URGDATA > 0
1644      uip_urghlen = (BUF->urgp[0] << 8) | BUF->urgp[1];
1645      if(uip_urghlen > uip_len) {
1646          /* There is more urgent data in the next segment to come. */
1647          uip_urghlen = uip_len;
1648      }
1649      uip_add_rcv_nxt(uip_urghlen);
1650      uip_len -= uip_urghlen;
1651      uip_urghdata = uip_appdata;
1652      uip_appdata += uip_urghlen;
1653  } else {
1654      uip_urghlen = 0;
1655  #else /* UIP_URGDATA > 0 */
1656      uip_appdata = ((char *)uip_appdata) + ((BUF->urgp[0] << 8) | BUF->urgp[1]);
1657      uip_len -= (BUF->urgp[0] << 8) | BUF->urgp[1];
1658  #endif /* UIP_URGDATA > 0 */
1659  }
1660

```

Figure 5 – CVE-2020-17437 (the source)

The root cause of the issue resides in the `uip_process()` function that handles incoming IPv4 packets. Figure 5 shows the code fragment that will be executed after a TCP handshake when a client sends TCP data to the stack. If the TCP Urgent flag is set (line 1642), and the stack is configured to handle the Urgent data (**UIP_URGDATA > 0**), the code fragment will get the Urgent offset from the packet

(**uip_urghlen**) and prepare the stack for receiving out-of-band data (lines 1644-1652).

However, by default, the stack is not configured to receive out-of-band data (the **UIP_URGDATA** constant is set to 0). Thus, upon receiving a TCP packet with the Urgent flag set, lines 1656 and 1657 will be executed instead. Here, the stack attempts to remove the out-of-band data from

TECHNICAL DIVE IN

the packet by moving the global incoming application data pointer (**uip_appdata**) past the Urgent data (line 1656) and adjusting the length of the TCP data by subtracting the Urgent data offset from the total TCP data length (line 1657). Here, it is crucial that **uip_appdata** points within a global fixed-size statically allocated buffer **uip_buf** since this buffer holds all incoming network packets.

Further, **uip_len** is passed into the function `uip_add_rcv_next()` that sets the “ack” number of the TCP ACK packet that the stack will send as a response. The global application data pointer **uip_appdata** is used by the `UIP_APPCALL()` callback (see Figure 6) for treating the incoming application data. By design, re-implementing this callback provides the ability to treat the application data in different ways (e.g., FTP, HTTP webserver or any other custom applications on top of the TCP protocol).

```

1666     if(uip_len > 0 && !(uip_connr->tcpstateflags & UIP_STOPPED)) {
1667         uip_flags |= UIP_NEWDATA;
1668         uip_add_rcv_nxt(uip_len);
1669     }
1670
1671
1672     if(uip_flags & (UIP_NEWDATA | UIP_ACKDATA)) {
1673         uip_slen = 0;
1674         UIP_APPCALL();
1675     }

```

Figure 6 – CVE-2020-17437 (the sinks)

At this point, the attackers control three things: (1) the global **uip_appdata** pointer, which will be set to point to the first TCP data byte after the TCP header; (2) **uip_len** – the length of the TCP data received by the stack; and (3) the Urgent data offset, which is taken directly from the TCP header (`(BUF->urgp[0] << 8) | BUF->urgp[1]`).

As none of these values are properly validated, attackers can craft TCP packets with arbitrary Urgent data offsets, achieving several side effects: (1) the **uip_appdata** pointer may be incremented by a large offset, pointing out-of-bounds of the packet (and even out-of-bounds of **uip_buf**); (2) **uip_len** is only two bytes wide; therefore, short packets with Urgent offset larger than it will cause the value of **uip_len** to overflow after the arithmetic operation at line 1657, which can lead to a denial of service on a device running this stack or have other consequences.

For example, if attackers send a packet with a small amount of TCP data (e.g., a single byte) and slightly larger Urgent data offset (e.g., 0x02), **uip_len** will overflow and become a large 2-byte value (0xffff). At the same time, **uip_appdata** will be advanced by the small Urgent offset (0x02), and it will still point within the range of **uip_buf**. This has the following side effect: The `uip_add_rcv_nxt()` function uses the **uip_len** value to increment the “ack” field of the response to be sent: The “ack” value of the incoming packet will be added to the resulting **uip_len** (see Figure 7).

TECHNICAL DIVE IN

Source	Destination	Protocol	Length	Info
172.18.0.1	172.18.0.2	TCP	54	8080 → 80 [SYN] Seq=0 Win=8192 Len=0
172.18.0.2	172.18.0.1	TCP	58	80 → 8080 [SYN, ACK] Seq=0 Ack=1 Win=1240 Len=0 MSS=1240
172.18.0.1	172.18.0.2	TCP	54	8080 → 80 [ACK] Seq=1 Ack=1 Win=8192 Len=0
172.18.0.1	172.18.0.2	TCP	55	8080 → 80 [PSH, ACK, URG] Seq=1 Ack=1 Win=8192 Urg=2 Len=1 [TCP segment of a reassembled PDU]
172.18.0.2	172.18.0.1	TCP	54	[TCP ACKed unseen segment] 80 → 8080 [FIN, ACK] Seq=1 Ack=65536 Win=1240 Len=0

Figure 7 – CVE-2020-17437 (Indirect information leak)

This implementation quirk essentially allows attackers to identify that this vulnerability exists without disrupting the operation of the device (indirect information leak).

If, on the other hand, we set a large Urgent data offset (e.g., 0xffff), the **uip_appdata** pointer will be pointing way past **uip_buf** (most likely at an invalid address), causing

memory corruption wherever the **uip_appdata** pointer is dereferenced. As noted above, the actual point at which the invalid pointer is accessed (and other potential impact vectors) depends on a specific implementation of the **UIP_APPCALL()** callback.

4.3. Common anti-patterns

Anti-patterns, also known as negative patterns, describe similar solutions to a common problem that may lead to negative consequences. In our case, an anti-pattern is a certain logic that is implemented in the same way in different stacks, thus leading to similar vulnerabilities.

By analyzing our sample of vulnerabilities (including AMNESIA:33), we understood that the most common anti-patterns come down to three bad development practices:

- A general **absence of basic bounds checks** and integer overflow checks.
- A **misinterpretation** or mis-implementation **of RFC documents** that define various protocols. Of course, at the same time, several aspects of specific RFCs are not strictly defined, leaving a large room for error (for instance, see the “Technical Dive In” example of CVE-2020-17443).

- A **heavy reliance on ‘shotgun parsing,’** which is the bad practice of mixing input validation and processing in a manner that facilitates the processing of only partially validated data.

From the analysis, it became clear that implementing the same protocols under similar constraints tends to produce similar bugs in similar places, providing vulnerability researchers with what is essentially a corpus of anti-patterns (i.e., similar known vulnerable pieces of code) and prioritized components on which to focus their efforts.

In AMNESIA:33 and the previous vulnerabilities, what stands out in common are generic issues with calculating and/or validating header and field lengths, properly parsing various header option fields, ensuring that there is enough data in the packet (in contrast to relying on what is specified in the header) and handling the notoriously vaguely specified TCP Urgent pointer. This is followed by issues with fragmentation reassembly, IP tunneling and request-reply matching.

HIGHLIGHTS

In relation to AMNESIA:33, **we have noticed the following anti-patterns not covered by previous research.**

DNS domain name length. When processing domain name entries, the parsing code must ensure that the value of the length byte of a domain name label corresponds to the number of bytes within the label. To spot a vulnerable implementation, it is often enough to find a loop (or a standalone function called within a loop) that processes domain names and has the following behavior:

- It takes the first byte of the label as the length variable and uses this length as the offset for advancing an internal packet data pointer without performing sufficient bounds checks.
- Additional checks should be carried out to ensure the maximum length of the label cannot be more than 63 bytes (e.g., 0xff is not a valid length of a label), and the maximum length of a domain name cannot be more than 255 bytes ([RFC 1035](#)). Also, only alphanumeric characters, digits and hyphens should be accepted as valid characters within a domain name ([RFC 1035](#)).

DNS domain name NULL termination. Most of the observations for the above anti-pattern apply here as well. Additionally, the vulnerable code may use a function that returns the length of a domain name within a DNS packet and expects that it is explicitly NULL-terminated (e.g., `strlen()` as in the “Technical Dive In” example of CVE-2020-25111). If this length is then used as an offset for memory operations without proper bounds checks, the code is most likely vulnerable.

IPv6 extension headers and options. Parsing of extension headers or specific options of an extension header (or all of them together) is typically done within a loop that incorporates a “switch” conditional statement. We made the following observations for vulnerable implementations:

- Variables that store the length of a specific extension header or the length of an individual option have either direct or indirect impact on the exit condition of the loop. These variables lack sufficient bounds checks to ensure that the data being parsed is within the packet limits and that the loop advances forward within the packet with each iteration. Sometimes, these variables can be 8 or 16 bytes long so that integer overflows may occur.
- Some implementations may contain flaws that are not relevant to memory corruption issues but can still have similar consequences like a successful Denial-of-Service attack. Therefore, it is worthwhile to spend more time to analyze whether the exit condition of the parsing loop can be abused (in a Technical Dive In we show the example of CVE-2020-17445, which allows attackers to either corrupt memory or achieve an infinite loop, depending on the input).

Relevant RFCs must be followed as strictly as possible. For example, the specific order of IPv6 extension headers should be maintained, and some headers, such as Hop-by-Hop extension, must not appear more than once within a packet ([RFC 2460](#)).

TECHNICAL DIVE IN

DNS domain name issues in AMNESIA:33

Figure 8 shows an example of how domain labels and names can be encoded within a DNS packet: Domain names may consist of one or multiple labels, where for each label the first byte represents the length of the label in bytes and the remaining bytes are alphanumeric characters of the label itself (some special characters are allowed as well). Labels can be chained into more complex domain names, but the very last byte of the domain name must always

be the NULL (0x00) byte, explicitly indicating where the domain name ends. For example, the domain name from Figure 8 starts with the byte 0x06 that indicates the length of the first label, followed by the bytes that correspond to the first label (0x67 0x6f 0x6f 0x67 0x6c 0x65 == "google"), continues with the length of the second label 0x03, the bytes that correspond to the second label (0x63 0x6f 0x6d == "com") and ends with the NULL terminator byte (0x00).

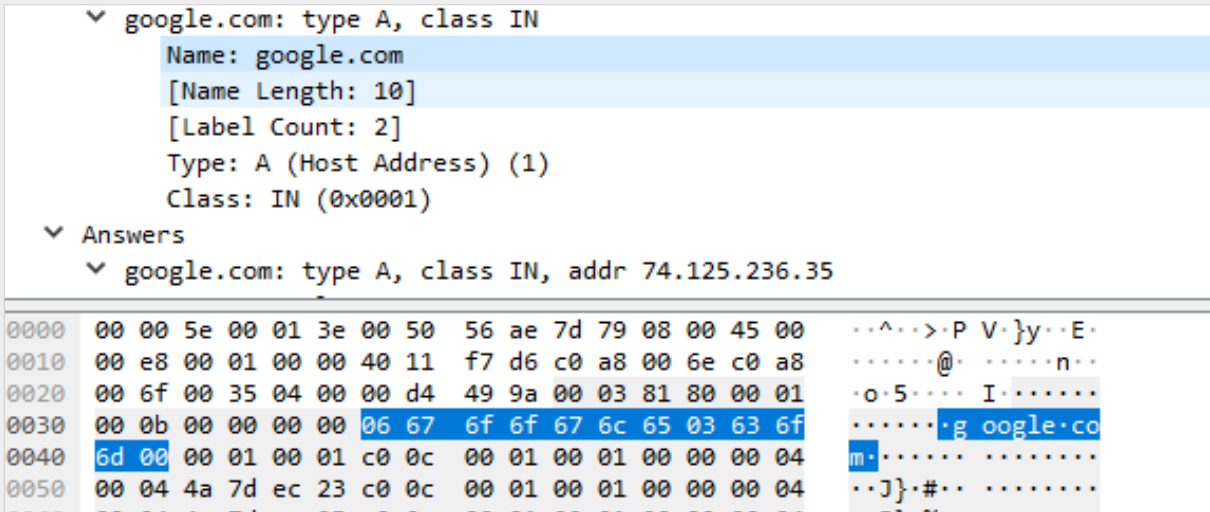


Figure 8 – An example of DNS domain label/name

One illustrative example for vulnerable DNS domain name parsing functions is related to CVE-2020-25111 that affects Nut/Net (the ScanName() function on Figure 9). Initially, cp is the pointer to the first byte of the domain name being parsed (i.e., the length byte of the first label), and *npp is the buffer into which the domain name is being copied while parsing. The code will read the total domain name length into the rc variable using the strlen() function (line 182) and allocate the *npp buffer based on rc (line 183). It then

will start parsing individual labels (lines 185-189) by first assigning the length of the first label to len and then copying labels into *npp byte by byte.

TECHNICAL DIVE IN

```
168 static uint16_t ScanName(uint8_t * cp, uint8_t ** npp)
169 {
170     uint8_t len;
171     uint16_t rc;
172     uint8_t *np;
173
174     if (*npp) {
175         free(*npp);
176         *npp = 0;
177     }
178
179     if ((*cp & 0xC0) == 0xC0)
180         return 2;
181
182     rc = strlen((char *) cp) + 1;
183     np = *npp = malloc(rc);
184     len = *cp++;
185     while (len) {
186         while (len--)
187             *np++ = *cp++;
188         if ((len = *cp++) != 0)
189             *np++ = '.';
190     }
191     *np = 0;
192
193     return rc;
194 }
```

Figure 9 – CVE-2020-25111

The ScanName() function lacks the necessary bounds checks:

1. The strlen() function (line 182) will return the amount of bytes in a sequence until the first NULL byte (0x00) is encountered. In this case, by specifying arbitrary sequences of bytes with the NULL byte placed at specific offsets of the packet (and outside of it), attackers can control the size of the heap-allocated ***npp** (line 183).
2. The length of a label (**len**) is taken directly from the packet and used as the “while” loop condition without any proper bounds checks. By setting arbitrary values to this length, attackers control the number of bytes written into the ***npp** buffer (up to 255 bytes).

To exploit the vulnerability, attackers may set arbitrary values to the label lengths (**len**), causing out-of-bounds writes past the domain name buffer (***npp**) and corrupting the memory. By carefully choosing a sequence of malformed domain name labels and placing NULL terminator bytes, attackers may have the ability to perform controlled OOB-W within the heap memory. This can lead to a remote code execution, as discussed in the Technical Dive In in Section 4.4.

Another category of issues related to processing DNS replies, identified within the AMNESIA:33 research, is when the number of response records specified in a DNS header of the reply packet does not correspond to the actual amount of response records available. A good example is CVE-2020-24334 that affects uIP.

TECHNICAL DIVE IN

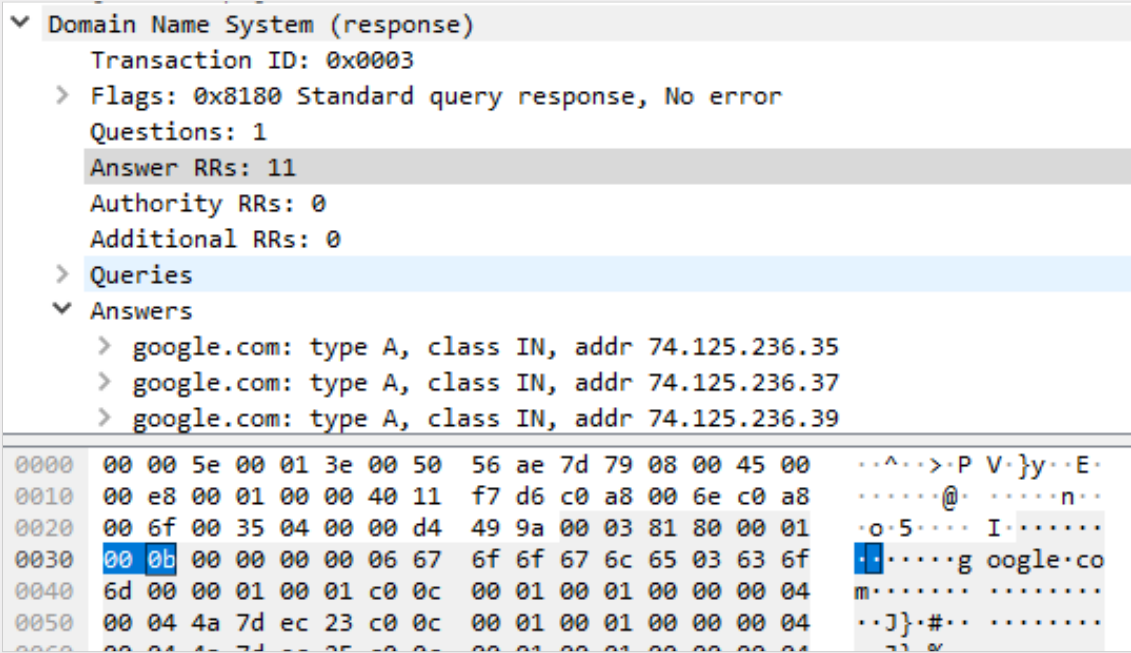


Figure 10 – An excerpt from a DNS response packet

Figure 10 shows an excerpt from a DNS response packet. The DNS header consists of a sequence of bytes starting with the transaction ID, flags, the number of questions, the number of response records, the numbers of authority and additional response records. After the DNS header, a packet

contains specific question record(s) (must be the exact number of question records specified in the header) and response record(s) (must me the exact number of response records specified in the header).

TECHNICAL DIVE IN

```

1 static void newdata(void)
2 {
3     uint8_t nquestions, nanswers;
4
5     int8_t i;
6
7     register struct namemap *namemapptr = NULL;
8
9     struct dns_answer *ans;
10
11     register struct dns_hdr const *hdr = (struct dns_hdr *)uip_appdata;
12
13     unsigned char *queryptr = (unsigned char *)hdr + sizeof(*hdr);
14
15     const uint8_t is_request = ((hdr->flags1 & ~1) == 0) && (hdr->flags2 == 0);
16
17     nquestions = (uint8_t) uip_ntohs(hdr->numquestions);
18     nanswers = (uint8_t) uip_ntohs(hdr->numanswers);
19
20     queryptr = (unsigned char *)hdr + sizeof(*hdr);
21     i = 0;
22
23
24     /* -----
25      * QUESTION PARSING CODE IS OMITTED FOR BREVITY
26      * -----*/
27
28     /* Answer parsing loop */
29     while(nanswers > 0) {
30         ans = (struct dns_answer *)skip_name(queryptr);
31
32         // [...]
33
34         skip_to_next_answer:
35         queryptr = (unsigned char *)skip_name(queryptr) + 10 + uip_htons(ans->len);
36         --nanswers;
37     }
38
39     // [...]
40
41 }

```

Figure 11 – CVE-2020-24334

Figure 11 illustrates an excerpt from the vulnerable DNS packet processing function related to CVE-2020-24334 (most of the code was omitted for brevity). Here, **hdr** is the pointer to the DNS header of the incoming DNS packet, taken from the global application data buffer **uip_appdata**; the numbers of questions and response records (**nquestions** and **nanswers**) are taken directly from the header (lines 17 and 18); the **queryptr** pointer points at the beginning of a resource record (initially it points at the first question record); and the “while” loop (line 29) iterates over the response records updating the **queryptr** so that it jumps to the next response record on the next iteration of the loop.

The problem here is that attackers have explicit control over the **nanswers** variable (it can be set directly in the DNS header; see the example at Figure 10), as well as the **queryptr** pointer. Therefore, if a packet with a large number of response records (e.g., 0xff) set in the DNS header and a smaller value of actual response records (e.g., < 0xff) is being processed, **queryptr** will eventually point out of the bounds of the packet (line 30), and OOB-R (potentially a Denial-of-Service) within the function `skip_name()` will happen.

TECHNICAL DIVE IN

IPv6 extension headers parsing in AMNESIA:33

We sketch the IPv6 extension headers processing vulnerabilities of AMNESIA:33 with one example: CVE-2020-17445 affecting PicoTCP.

```
1 static int pico_ipv6_process_destopt(struct pico_ipv6_exthdr *destopt, struct pico_frame *f, uint32_t opt_ptr)
2 {
3     uint8_t *option = NULL;
4     uint8_t len = 0, optlen = 0;
5     opt_ptr += (uint32_t)(2u);
6     option = ((uint8_t *)&destopt->ext.destopt) + sizeof(struct destopt_s);
7     len = (uint8_t)((destopt->ext.destopt.len + 1) << 3) - 2;
8     while (len) {
9         optlen = (uint8_t)((option + 1) + 2);
10        switch (*option)
11        {
12            case PICO_IPV6_EXTHDR_OPT_PAD1:
13                break;
14
15            case PICO_IPV6_EXTHDR_OPT_PADN:
16                break;
17
18            case PICO_IPV6_EXTHDR_OPT_SRCADDR:
19                break;
20
21            default:
22                // THE CODE HERE IS OMITTED FOR BREVITY
23                break;
24        }
25        opt_ptr += optlen;
26        option += optlen;
27        len = (uint8_t)(len - optlen);
28    }
29    return 0;
30 }
```

Figure 12 – CVE-2020-17445

Figure 12 shows the vulnerable function for processing the IPv6 Destination Options extension header (CVE-2020-17445). This function parses the options present in this extension header one at a time. The **option** pointer points at the current option being parsed; the **len** variable initially contains the length of the extension header (and then is used to track the number of bytes being parsed, being the exit condition of the “while” loop); and the **optlen** variable reads the length of the current option being parsed.

The main issues of CVE-2020-17445 are the following:

- Attackers can explicitly control **optlen** by setting arbitrary length of an option within the Destination Options extension header, and there are no sanity checks for the value of this variable.
- Attackers can implicitly control **len** (line 27) that is used within the exit condition of the “while” loop (line 8), and there are no sanity checks for the value of this variable.

TECHNICAL DIVE IN

- Attackers can implicitly control the **option** pointer (line 26), and there are no bounds checks ensuring that this pointer is pointing to the data within the packet being processed.

Thus, by sending a carefully crafted packet, attackers can achieve OOB-R (potentially a Denial-of-Service) by shifting the **option** pointer into the unmapped memory. With this degree of control, attackers may also cause a “silent”

Denial-of-Service: The “while” loop will never terminate, and the stack will not process other incoming packets. The **optlen** variable is an unsigned 8-bit integer; therefore, it may overflow after the arithmetic operation at the line 9. If attackers manage to cause an overflow such that **optlen** becomes 0x00 at line 9 (i.e., by setting an option length to 0xfe), the values of **option** and **len** will not change at the lines 26 and 27, and the loop will iterate indefinitely.

4.4. What about exploitability?

Whether a vulnerability in a protocol stack is exploitable on any actual device depends on [many factors](#), such as configuration settings, target platform, the presence of exploit mitigations and the freedom the attacker has in shaping the target’s memory and state.

It is well-known that embedded systems – IoT and OT devices – usually lack the hardware, software or resources required to deploy [modern exploit mitigations](#), such as [non-executable data memory](#) (also known as ESP, DEP, NX and W^X), [address space layout randomization](#) (ASLR) and [stack canaries](#) for protection against memory corruption exploitation.

The RTOSes that typically run on embedded systems rarely offer appropriate memory segmentation and privilege separation ‘out of the box.’ Thus, application, networking and OS code often all run in the same flat address space.

This combined lack of exploit mitigations and memory protection in embedded systems tends to render exploitation significantly easier than on modern IT devices, such as servers or laptops, thus increasing the risk posed by issues on these systems.

This also means that the impact of a vulnerability will manifest differently on different devices. During

our study, **we performed a thorough analysis of the vulnerabilities described in Table 7 to understand their exploitability and potential impact. We found that exploitability is influenced heavily by the following factors:**

- Stack configuration:** TCP/IP stacks are highly configurable, allowing for enabling and disabling various substacks, specifying buffer sizes, selecting different kinds of memory allocators, regulating interaction with network drivers and handling debugging functionality. For example, we found some bounds checks implemented as part of assertion predicates, which are often turned off in production, so the exploitability of some issues depends on the assertion configuration.
- Networking Hardware & Driver:** TCP/IP stacks typically ‘talk’ to network interface abstraction code, which in turn talks to a NIC (or MAC) driver to translate between the specifics of a piece of networking hardware and a generic API. TCP/IP stacks often can be configured to offload packet validation and filtering, and certain network controllers do so autonomously regardless of stack configuration. Depending on the nature of a vulnerability, this can influence whether a malicious packet ever gets to reach the code it seeks to exploit.

- **Target Platform:** In some cases, exploitability of an issue is highly dependent on the target's hardware architecture and configuration. For example, [CVE-2018-16524](#) affects the FreeRTOS+TCP stack by allowing an attacker to provide an MSS value of 0 and cause a division-by-zero, which can lead to a DoS. However, the handling of division-by-zero depends on the target platform, and in some flavors of ARM, the division can return a value of 0 and not an exception, thus rendering the vulnerability unexploitable.

It is crucial to keep in mind that a device that uses a particular IP stack will not automatically be exploited. Even when a vulnerability on a device can be exploited, the impact of a vulnerability varies greatly.

TECHNICAL DIVE IN

Exploiting CVE-2020-25111

CVE-2020-25111 is a classic heap buffer overflow occurring during the processing of the name field of a DNS response resource record. An attacker can control the size of the allocated buffer while writing an arbitrary number of bytes to it, allowing the attacker to corrupt adjacent memory, including metadata of other heap nodes. Nut/OS (the OS that runs the Nut/Net stack) uses a single, non-segregated, singly linked free-list in combination with a deterministic, best fit, address ordered allocation algorithm which performs forward coalescing. Heap guards are optional and static. Heap nodes consist of a metadata structure followed by the data itself, as follows:

```
struct _HEAPNODE {
    size_t hn_size;      /* Size of this node. */
    HEAPNODE *hn_next; /* Link to next free
                        node. */
};
HEAPNODE *heapFreeList;
```

To exploit CVE-2020-25111, an attacker can abuse the fact that Nut/Net's DNS component allocates and frees resource record fields of sequential answers on the heap and uses this for granular heap shaping. Consider the following Nut/Net code, together with the function `ScanName` that was shown in Figure 9:

```
static uint16_t ScanBinary(uint8_t * cp, uint8_t
** npp, uint16_t len) {
    if (*npp)
        free(*npp);
    *npp = malloc(len);
    memcpy(*npp, cp, len);
    return len;
}

static uint16_t DecodeDnsResource(DNSRESOURCE *
dor, uint8_t * buf) {
    uint16_t rc;
    rc = ScanName(buf, &dor->dor_name);
    rc += ScanShort(buf + rc, &dor->dor_type);
    rc += ScanShort(buf + rc, &dor->dor_class);
    rc += ScanLong(buf + rc, &dor->dor_ttl);
    rc += ScanShort(buf + rc, &dor->dor_len);
    rc += ScanBinary(buf + rc, &dor->dor_data,
dor->dor_len);
    return rc;
}

...
for (n = 1; n <= (int) doh->doh_answers; n++) {
    dor = CreateDnsResource(dor);
    len += DecodeDnsResource(dor, pkt + len);
    if (dor->dor_type == 1)
        break;
}
```

TECHNICAL DIVE IN

Here, an attacker can send a DNS response with two answers. The first answer's resource record has a name and data field with carefully chosen sizes to ensure beneficial allocation and create a situation where the node holding `dor->dor_name` directly precedes the node holding `dor->dor_data` which in turn is followed directly by a free node. Due to reasons that are out of scope for this report, we require an allocated node in between our target-free node and our overflowed node when the overflow happens.

The second answer has a resource record with a malicious name, causing an allocation of a size identical to the first answer's name size. Since that node was just freed, the best-fit allocator will allocate this name at the exact same position, right in front of the still allocated `dor->dor_name` and subsequent free node. Then the overflow happens, and we overwrite both the allocated node and the metadata of the free node. When the code starts processing the data field, it will deallocate the (now corrupted) previous node and allocate a new one of a size we control. The allocator will walk the free list until it encounters our corrupted node whose `hn_next` field will redirect the allocator to a target memory area of our choice. If the `size_t` value located there matches the requested allocation size of our data, the allocator will think it has found a best fit 'node' and 'allocate' the new `dor->dor_data` at an address of our choice. Subsequently, the code will copy a number of bytes under our control to that location.

We can abuse this controlled allocation to form a write primitive that we can use to corrupt data or code of interest and get RCE. In our case, we used it to overwrite the local `stackframe` metadata (including the saved return address) of `NutDnsGetResource` parent function so that upon function return, we hijack control-flow. There are a few limitations on how we have to craft our malicious DNS reply, but the most significant one is that the `size_t` value

located at our target memory during controlled allocation needs to be both known to us and present a reasonable size. Luckily, `NutDnsGetResource` has a few local stack variables that are suitable for this purpose, including `raddr`, which will be set to the IP source address of our malicious packet as a value we control and can set to anything we like.

Using the above materials, we can create an exploit that redirects control-flow to a ROP chain that will ensure cache coherency on architectures that require it (e.g., ARM, MIPS) and finally shellcode of our choice. Exploitability of this vulnerability is determined by a few factors:

1. Whether the DNS component is enabled and used
2. Target platform and architecture (e.g., memory protection and organization, word sizes, etc.)
3. Baseline heap activity intensity
4. Heap allocator used
5. Exploit mitigations used (e.g., NX, ASLR, etc.)

Factor 3 can be mitigated by having more extensive heap shaping activity using additional DNS answers. Factor 4 is most likely irrelevant in practice since Nut/OS seems to heavily favor using its own allocator which we described briefly above. Factor 5 is irrelevant in practice since Nut/OS has no mitigation support. (It does not even have regular memory protection support.)

HIGHLIGHTS

The impact of memory protection on TCP/IP stack vulnerabilities

The common lack of support for memory protection or segmentation in embedded devices means that some OOB-R and OOB-W vulnerabilities might not cause a crash immediately since there is no protected or (un) mapped memory that would cause a fault handler to be invoked. That does not mean that a crash cannot occur, but it is more likely that this will be very unpredictable, for example by corrupting some piece of memory that at a later point in time gets used by the program in a way that causes unexpected behavior or by reading from or writing to memory mapped peripherals causing unexpected interactions with other system components.

The absence of memory protection and segmentation, while making devices more vulnerable, ironically also renders some impacts less reliable. On the other hand, it does mean that an exploited device will not shut down, hang or reboot 'safely' and in a controlled way through a fault handler.

This is another complicating factor in determining impact, since on one device there might be memory protection support while on another there might not be, adding a lot of nuance.

HIGHLIGHTS

A side note on fuzzing

Besides the impact on exploitability, the lack of memory protection may help to explain why some of these vulnerabilities have not been reported before. If someone fuzzes a device in a black-box fashion over the network or using an emulated image, then they are unlikely to trigger a crash soon after a test case is sent, which makes it seem

like there is no issue, while in reality the corruption is now latent in the device. Regular penetration tests, stress tests and device standards compliance tests (e.g., [IEC 62443-4](#)) are likely to take place in such a black-box fashion and conclude there is no issue. Since we fuzzed the source-code in a memory protected environment, this allowed us to tightly couple test cases to crashes.

4.5. What is the actual danger?

When publishing vulnerability advisories, researchers and vendors try to define the impact type of a vulnerability, namely the type of harm an attack could cause if the vulnerability were exploited. Figure 13 gives a breakdown of the vulnerabilities in our sample by these potential

impacts, but this requires some explanation. Figure 13 shows what we call the "immediate impact," namely the one **assigned to a vulnerability based on a researcher's judgement**.

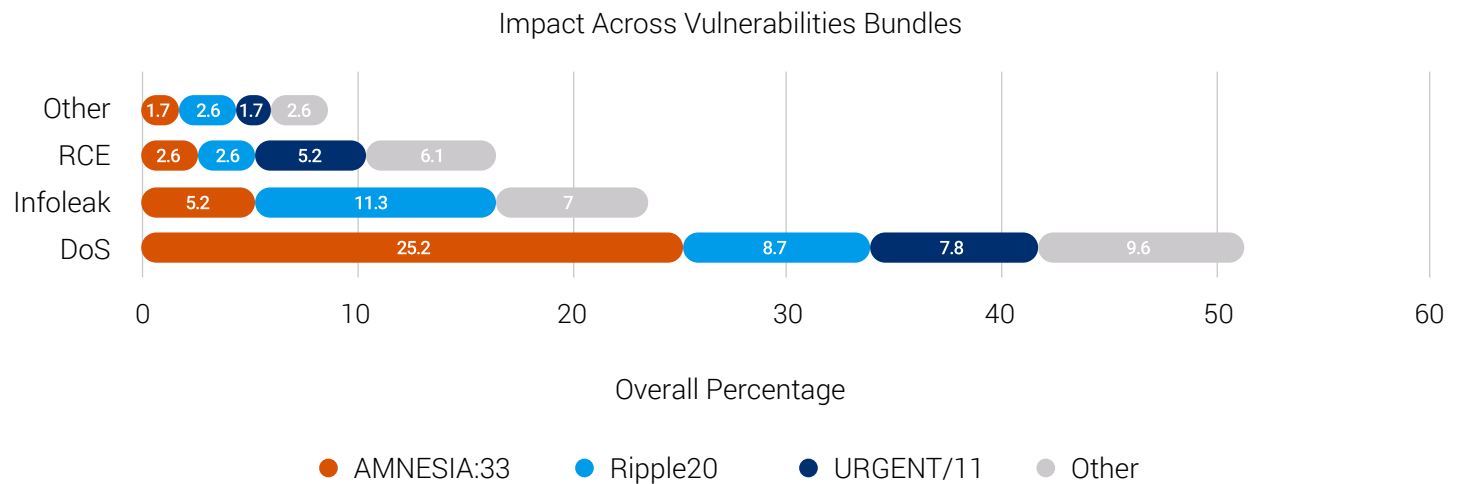


Figure 13 – Vulnerability impact categories within popular TCP/IP stacks

However, only relying on the immediate impact can lead to missing the real potential of a vulnerability. For instance, an out-of-bounds read that is associated by the researchers to an information leak also could be turned into a denial-of-service if the attacker attempts to read from unmapped or protected memory to exploit it, hence causing a segmentation fault. Similarly, a vulnerability reported as a denial-of-service may be turned into remote code execution under the right circumstances.

This means that the real impact is highly contextual. For example, an information leak that discloses a few bytes from memory might not have [Heartbleed](#) implications, but it could very well become part of a larger exploit chain with far more significant impact than the sum of its parts.

Another factor to consider when discussing the real impact of a vulnerability is what component it affects. RCE in the IP stack is different from RCE in applications (e.g., DNS or HTTP) since in the former case the target does not need to be listening actively on any port for the packet to be fully processed. **As a result, RCEs on IP stacks are far more dangerous since they have the potential to breach firewalled and hardened hosts.**

Finally, another key factor for which to account is what type of device runs the vulnerable code. **For example, denial-of-service is often considered significantly less important than remote code execution, but this is not the case in critical OT environments where availability is crucial.** RCEs in critical embedded devices can be used to commit fraud in a smart meter, breach corporate networks via [building automation](#) and [routers, VPNs, firewalls or gateways](#), or attempt to [cause physical damage on a safety controller](#).

To disrupt or damage a critical operation, an attacker could leverage a vulnerability in the TCP/IP stack of a PLC controlling the opening and closing of a dam. For the attack to be successful, the PLC needs to run the vulnerable component of the stack in the right hardware component. Indeed, in certain architectures, PLCs integrate Ethernet communications on the same CPU as the processor module, meaning that the attacker can easily reach the processor via an RCE on the Ethernet and take control of the device. However, in more modular architectures, Ethernet communication may be a separate module with its own CPU, making it harder for the attacker to get complete control of the

PLC by exploiting the RCE in the Ethernet module. Once again, **this shows how the real impact of a vulnerability is heavily dependent on the context surrounding the targeted device.**

This is something asset owners and network operators must keep in mind when assessing the impact a vulnerability can have in their environment. The real impact can be indeed very different from the assigned CVSS score, for good or for bad. In this regard, a help for asset owners comes from the [Environmental Metrics](#) of CVSS, which allow one to customize a vulnerability's score based on the importance of an asset in the organization.

5. Estimating the reach of AMNESIA:33

5.1. Where you can see AMNESIA:33 – the modern supply chain

Figure 14 shows a few examples of components and devices that we identified running the vulnerable stacks. The AMNESIA:33 vulnerabilities can be found in products that range from embedded components (such as Systems on a Chip – SoCs, connectivity modules and OEM boards) to consumer IoT (such as smart plugs and smart thermostats), and from networking and office equipment (such as printers, switches and server software) to OT (such as access control devices, IP Cameras, RTUs and HVAC).

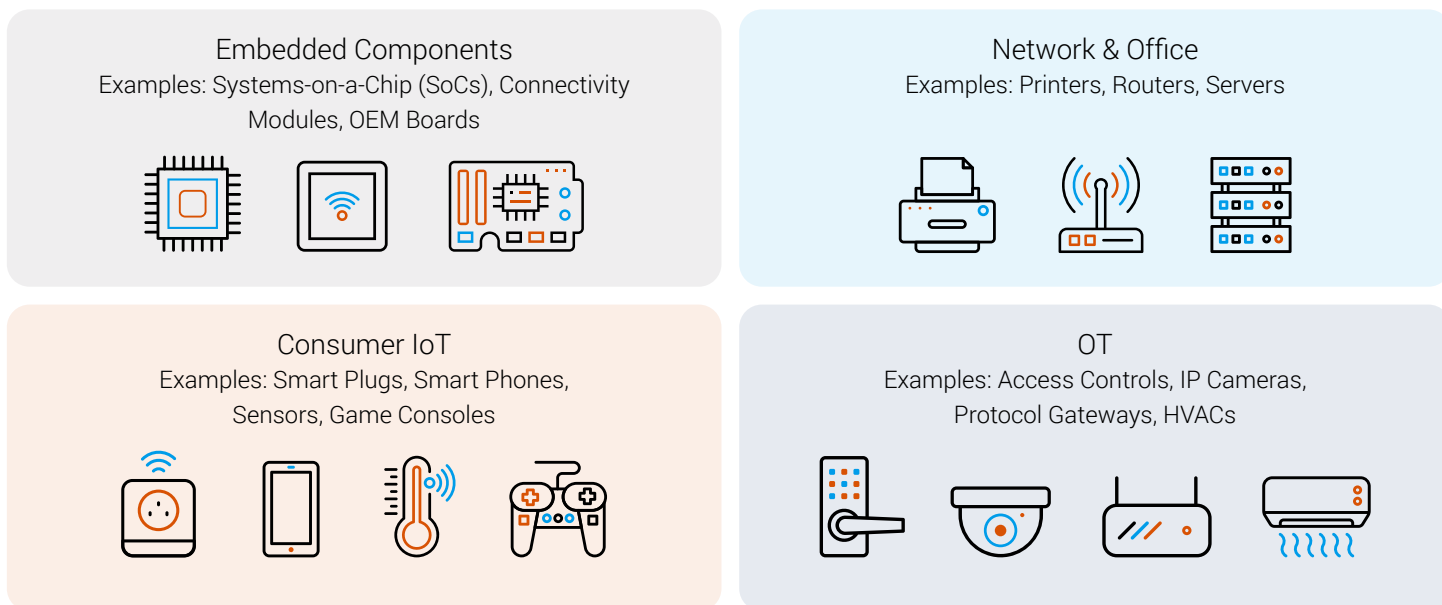


Figure 14 – Examples of components and devices running the vulnerable stacks

To understand the real reach of AMNESIA:33, we first need to understand how embedded devices are composed. Typically, IoT, OT and even IT devices found in enterprise and home networks are built up from several hardware and software components, including:

- Microcontroller Units (MCUs), which are very small computers in single microchips.
- Dedicated modules, which are hardware components that provide specific functions, such as Wi-Fi or USB connectivity.
- Systems-on-a-Chip (SoCs), which are microprocessors with a number of integrated peripherals on the same chip.
- Original Equipment Manufacturer (OEM) boards, which provide ready-to-go system boards to be used in the product of another manufacturer.

These components come from a device vendor's supply chain (i.e., they are produced by other software and hardware component vendors, and they are **"mixed and**

matched" based on different design constraints (such as specific lightweight TCP/IP stacks needed for low-energy or low-memory consumption in wireless sensors). Each of these components runs embedded software that may include a TCP/IP stack.

It is seldom the case that the end user of a device has complete knowledge of all the hardware and software components that are present on it – known as a Bill of Materials, or BOM. On the contrary, it is often a surprise to see how many and which components eventually enter in the final product. For example, Figure 15 shows the components of a Broadlink Smart Plug. The plug contains the [MediaTek MT7681](#) a popular Wi-Fi module that leverages the vulnerable stack uIP. There are several SoCs that are based on the MT7681, such as the [To-Link TMA1507A](#), the [HiLink HLK-M30](#) and [HLK-M35](#), the Scinan [SNIOT505](#), the Ogemray [GWF-KM22](#) and the Broadlink [WT1SBS](#), [WT1SBSL](#) and [WT1FBS](#). If we look at devices using Broadlink SoCs, we have examples such as the SP mini and MP1 smart plugs.



Figure 15 – Supply chain example, the Broadlink Smart Plug

HIGHLIGHTS

In our study, we found that several Wi-Fi modules rely on uIP, probably due to its very small memory footprint.

Another example, highlighting the security problems arising with long supply chains, is illustrated in Figure 16, which is anonymized because of the criticality of the assets involved. We found Vendor A of UPS devices that relies on another Vendor B for its network management cards. Vendor B, in turn, integrates an embedded RTOS from Vendor C on these network managements cards. Finally, the embedded RTOS includes a vulnerable

TCP/IP stack from Vendor D. The problem arises when Vendor C goes out of business (as it actually happened) and the RTOS for the network management card is no longer supported. This means that even if the uIP stack is patched, this patch will not become part of the RTOS distribution or the network management card, leaving the UPS un-patchable.

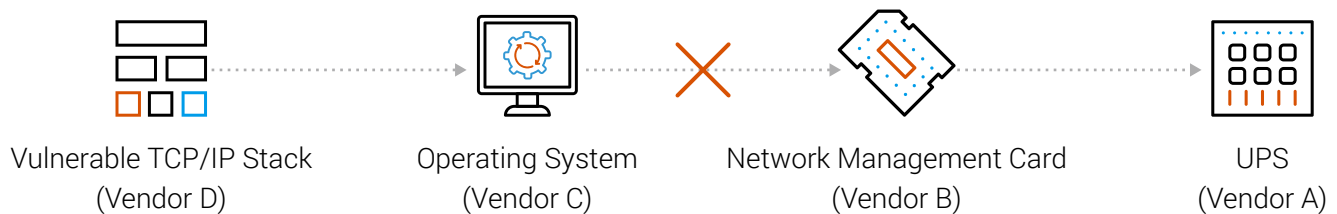


Figure 16 – Supply chain example, security issues on a UPS

5.2. The challenge – identifying and patching affected devices

Ripple20 and URGENT/11 taught the community that at the time of disclosure, it is difficult to understand the real reach of vulnerabilities affecting TCP/IP stacks.

For example, Schneider Electric has released a [security bulletin](#) in response to URGENT/11 confirming that over 60 device series have been found affected, and this bulletin still gets regular updates even almost a year after the original disclosure of URGENT/11. Cisco in its [original response to Ripple20](#) listed some of its devices as vulnerable but recognized that there were still devices under investigation. Some vendors might remain unaware whether their products are affected for a long time after the original vulnerability disclosure. For instance, the vulnerable version of the IPNet stack is quite old (released circa 2006-2007), and it since has been integrated into many products of many vendors.

Yet, after almost a year since URGENT/11 and a few months since Ripple20, both sets of vulnerabilities

are thought to affect hundreds of millions of devices, including categories of devices that clearly underline the criticality of embedded TCP/IP stacks. When we investigated Forescout's Device Cloud, we saw more than 32,000 instances of [IPNet \(on VxWorks\) in July, 2019](#) and more than 90,000 instances of [Treck in June, 2020](#) using signatures such as OS classification, application banners and DHCP request fingerprinting. These product categories include industrial controllers from ABB, Siemens, Schneider Electric, Rockwell Automation and others; healthcare systems from Philips, GE, Baxter and others; networking equipment from Cisco, SonicWall and others; as well as enterprise devices such as printers and VoIP phones from HP, Alcatel-Lucent and others.

AMNESIA:33 affects multiple stacks that are not owned by a single company but maintained as open source projects. After their code is published on a collaborative repository, such as GitHub or SourceForge, new forks and the appearance of a variety of versions of the code are almost inevitable.

The risk is that these vulnerabilities can spread easily and silently across multiple codebases, development teams, companies and products since these stacks form the basis of other software, operating systems, SoCs, embedded modules and development boards used to create a multitude of devices. This can happen because of the integration of faulty code in a project or because a new project starts as a fork of a vulnerable one. Below, we describe some of these situations that we observed with AMNESIA:33.

- uIP started as a standalone project, which then became part of the Contiki OS. Contiki became a popular operating system for the Internet of Things, and then was branched into Contiki-NG. Contiki also forms the basis of the [Thingsquare IoT platform](#), which is [used by](#) companies such as ABB and Electrolux. Some of these versions of uIP, such as Contiki-NG and Thingsquare, are still maintained and some are not, such as the original uIP 1.0, but they are all still available for download and use.
- The NuttX RTOS started by importing uIP, but then evolved its code independently. We could verify that at least one vulnerability still applies to NuttX. Similarly, parts of NuttX code can be found on Samsung's [TizenRT RTOS](#) and the [micro-ROS](#) robotics OS.
- The [open-iscsi](#) project, which provides an implementation of the iSCSI protocol used by Linux distributions, such as Red Hat, Fedora, SUSE and Debian, also imports part of the uIP code. Again, we were able to detect that some CVEs apply to it.
- The [u-boot_mod](#) project is a modification of UBoot 1.1.4 for routers that includes, among other things, a [web server based on uIP 0.9](#). This was based on D-Link firmware that also includes a webserver based on uIP 0.9. We saw at least the D-Link DIR 505 router running it.
- The mDNS component of the [nanostack](#) used by the ARM mbed OS is a copy of FNET's mDNS component,

so CVE-2020-24383 applies to this new stack. We also found that Zephyr RTOS' stack derives part of its [TCP handling code](#) from FNET, but in this case, we did not detect any vulnerability spreading.

Open source code should make it easier to fix vulnerabilities. Ideally, when a new vulnerability is disclosed, any member of the project could prepare a security patch. However, during this research, we discovered that because of the many forks, branches and unsupported yet-available versions, it is difficult to get these patches applied everywhere.

We contacted the [ICS-CERT](#) and the [CERT Coordination Center](#) to help in the disclosure, patching and vendor communication for the AMNESIA:33 vulnerabilities. They in turn got the help of [GitHub's security team](#) to find and contact affected repositories. Despite much effort from all the parties, official patches were only issued by the Contiki-NG, PicoTCP-NG, FNET and Nut/Net projects. At the time of writing, no official patches have been issued for the original uIP, Contiki and PicoTCP projects, which we believe have reached end-of-life status but are still available for download. Some of the vendors and projects using these original stacks, such as open-iscsi, issued their own patches.

5.3. Facing the challenge – estimating numbers

Identifying affected devices is even harder for open source stacks because their code can be reused easily and adapted across many projects. This deliberate fragmentation makes it much more challenging to account for the presence of a vulnerable component in a device.

To have an initial idea of the types, vendors and number of devices impacted by these new vulnerabilities, we looked at three data sources:

- 1. Open source intelligence:** We looked at product documentation, datasheets and licensing information that often mention open source components used by a device. For instance, the [datasheet](#) for some ICS controllers in the [Siemens SIRIUS](#) line mentions uIP. As another example, Netgear makes available the [GPL code they use in their products](#). By looking at those, we could identify that some switch models run the Contiki OS.
- 2. Online devices:** We queried [Shodan](#), [Censys](#) and [Fofa](#) for devices having banners that indicate the use of the stacks (e.g., “Server: uIP/1.0” or “Server: uIP/0.9” for uIP). Interestingly, Fofa has predefined tags for some of the OSe or IP stacks, including app=“Thingsquare-Contiki” for Contiki, app=“FNET-HTTP-Freescale-Embedded-Web-Server” for FNET and app=“Ethernut-Project” for Nut/Net. Figure 17 shows the result of a search for “Contiki” on Shodan.

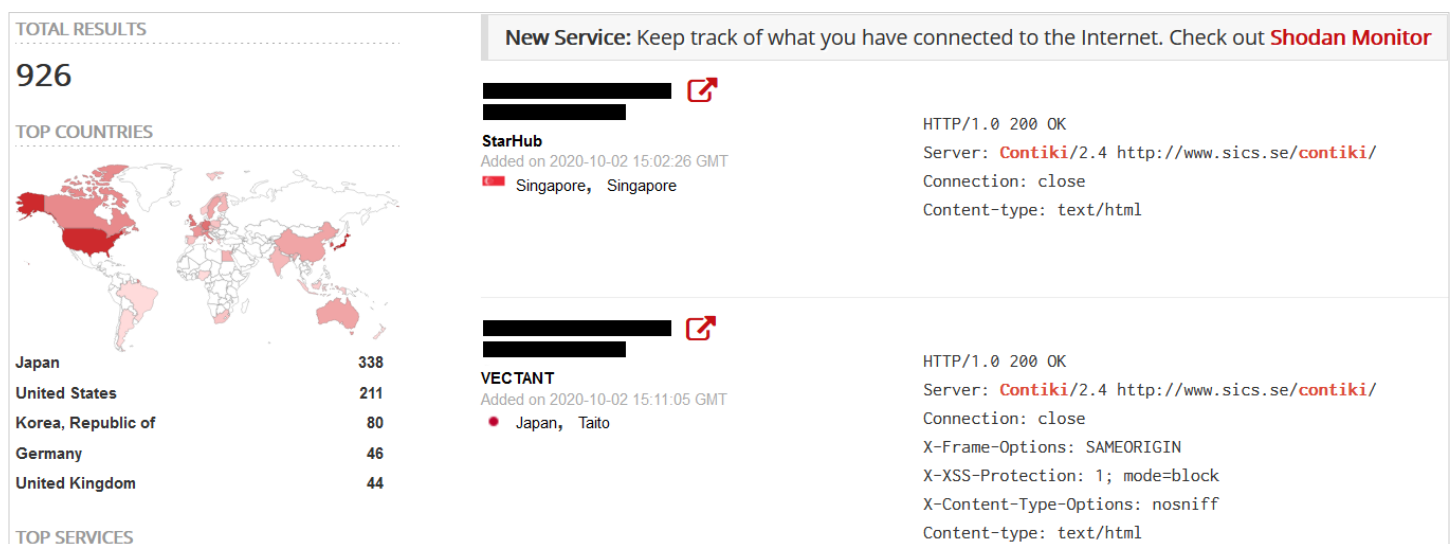


Figure 17 – Query for “Contiki” on Shodan

- 3. Forescout Device Cloud:** Device Cloud is a closed repository of information coming from devices monitored by Forescout appliances. We queried it for information such as OS classification, application banners and DHCP request fingerprinting, similar to [what was done for Ripple20](#).

Although we put a lot of effort into identifying reliable sources of information, all three sources mentioned above might occasionally lead to false positives. For instance, datasheets and product licenses may mention a component that is not used by a specific device (e.g., FreeRTOS ships with uIP and lwIP code, but either one,

the other or a third stack is used, while the license may mention both), and online devices may have an HTTP server of one stack and then use another stack for other protocols.

The main issue, however, lies with false negatives: We may not have enough information to identify the use of a stack on a device. For instance, in case the usage is not mentioned in the documentation, the device doesn't have an application-layer banner, and it is not present on a Forescout customer. **Therefore, we expect the numbers below to underestimate the actual numbers of vendors and devices.**

After analyzing the data from the three sources aforementioned, we compiled a list of more than 150 unique potentially affected vendors and device models. We also estimated the number of device units vulnerable in the wild in the order of millions.

Below, we present some details of this analysis. We do not mention here vulnerable vendors or devices by name because investigations are still ongoing, but we present statistics about vendors, device types and device units.

5.3.1. How many vendors

Table 8 shows the number of vendors we identified using each stack. Notice that the “total unique” is not the sum of the rows for each stack because some vendors use more than one stack, and we only count them once. Vendors are divided in two: “component vendors” are those that sell RTOS, IoT stacks, MCUs, SoCs and other components used to create end consumer or enterprise devices, and “device vendors” are those that sell end devices directly to consumers or companies.

Table 8 – Vendors identified

Stack	Total vendors	Component vendors	Device vendors
uIP	125	26	99
Nut/Net	24	1	23
picoTCP	10	8	2
FNET	5	2	3
Total unique	158	36	122

5.3.2. What device types

Figure 18 shows a division, in macro categories, of the potentially vulnerable device models identified from the three data sources. The largest category is **IoT, both enterprise and consumer**, which includes devices such as cameras, environmental sensors (e.g., temperature, humidity), smart lights, smart plugs, barcode readers, specialized printers, audio systems for retail and a few healthcare devices. IoT is followed by **OT equipment for Building Automation Systems**, which includes devices such as physical access control, fire and smoke alarms, energy meters, batteries and HVAC systems. Then we have **OT equipment for Industrial Control Systems**, which

includes devices such as PLCs, RTUs, protocol gateways and serial-to-ethernet gateways. **Following is IT**, which includes devices such as printers, switches and wireless access points.

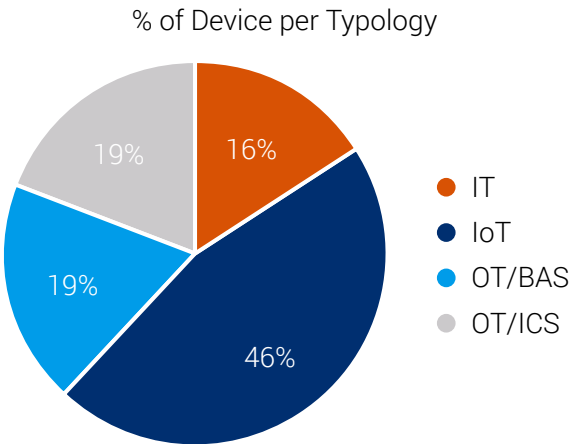


Figure 18 – Device type distribution

5.3.3. How many device units

Estimating the number of existing individual vulnerable device units is the most difficult task because some devices, such as PLCs, RTUs and other OT equipment, are known to be very popular but are rarely found online since they are not supposed to be internet-connected. Besides, device components are not always advertised in documentation or in network traffic, although popular SoCs are shipped by manufacturers in the order of millions per quarter.

Nevertheless, we found around 11,000 online instances of potentially vulnerable devices and more than 35,000 instances on Device Cloud. Drilling down into the Device Cloud data, Figure 19 shows a distribution of the potentially vulnerable devices per industry vertical where they are deployed. The Figure shows that **government, healthcare, services and manufacturing are the verticals with the highest number of potentially vulnerable devices.**

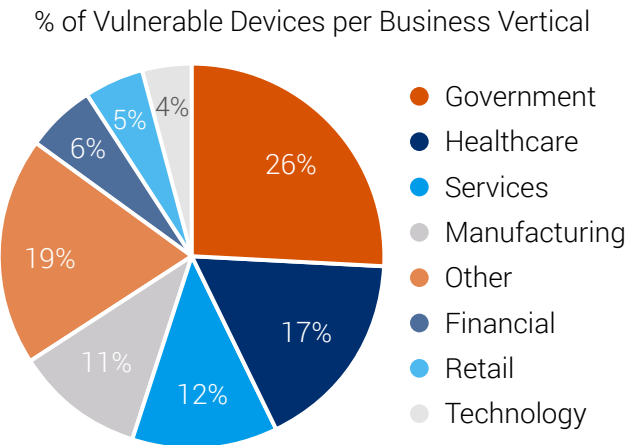


Figure 19 – Potentially vulnerable devices per industry vertical

We also extended this analysis by looking at marketing material about some specific device models, including case studies from vendors and market analysis from independent firms. Table 9 shows a breakdown of these data sources and the number of devices running the vulnerable stacks that we identified. Note that no public

information related to the number of units running the FNET stack has been found at the time of writing.

As with the previous numbers, the figures below are an underestimation because the marketing material was only available for a very limited number of devices.

Table 9 – Estimation of vulnerable devices, a breakdown

Stack	Device Type	Device Units (~)	Source
uIP	IoT – Wi-Fi SoCs	10M	Market Analysis
	IT – Switches	5M	Market Analysis
	OT/BAS – Fire Control Panels	45k	Case Study
Nut/Net	OT/BAS – Temperature Sensors	13K	Case Study
picoTCP	OT/BAS – HVAC	100k	Case Study
	OT/ICS – RTUs	200k	Case Study

6. An attack scenario

Figure 20 shows a simplified, though realistic, network configuration for a typical enterprise, and it will be used to discuss how AMNESIA:33 could be exploited by a malicious actor to damage the enterprise.

In our example, the enterprise has four locations: a retail branch, a home office, an enterprise HQ and a sub-station. To facilitate our discussion, we will ignore the presence of internal network segmentation. However, to better reflect the reality, we assume that while the retail branch, the home office and the enterprise HQ are internet-connected, the sub-station is isolated and can only be accessed from within the Enterprise HQ network.

Note that all the device types in Figure 20 have at least one instance that is vulnerable to AMNESIA:33. Namely, we found at least one device model that runs one of the vulnerable stacks.

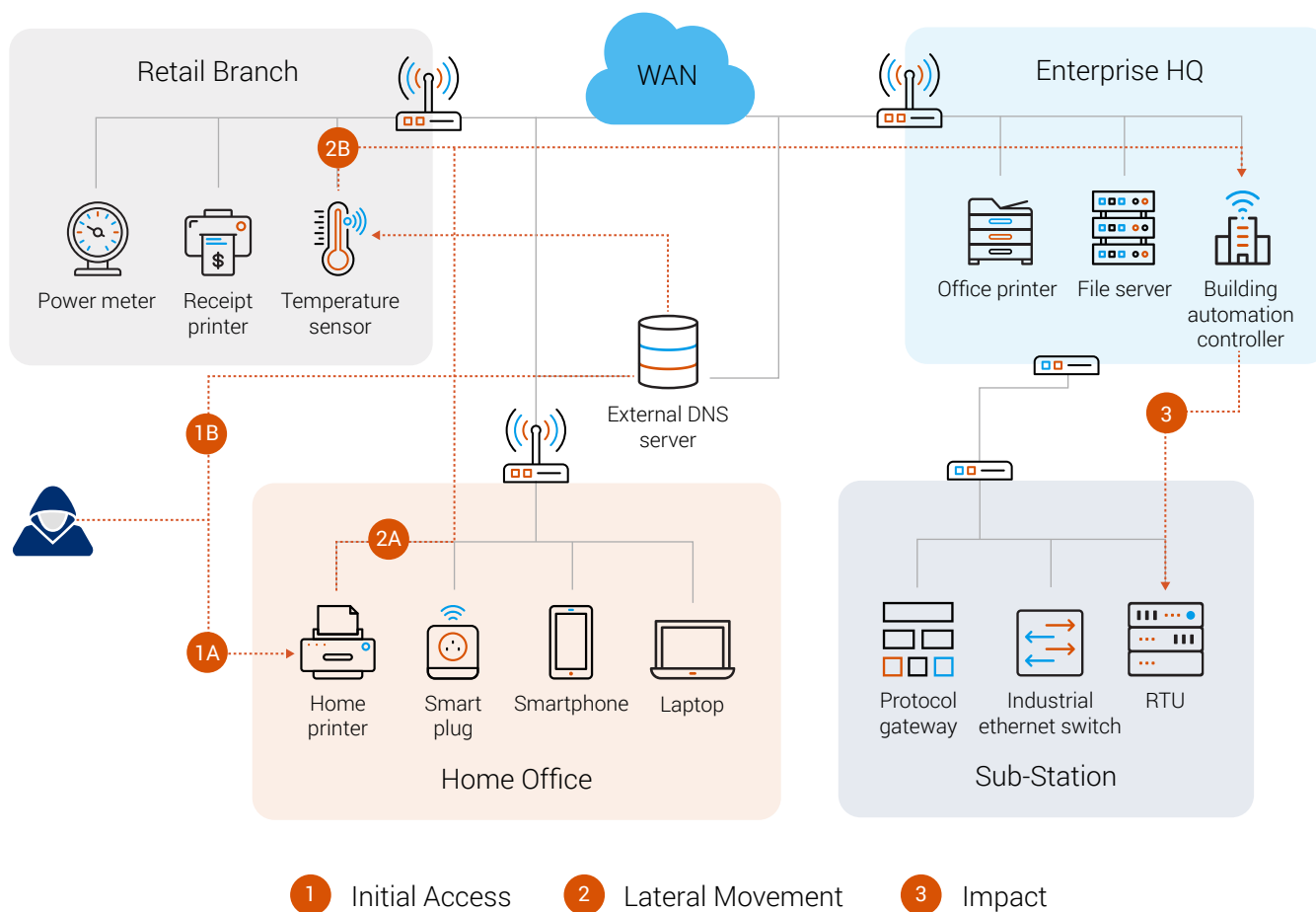


Figure 20 – How AMNESIA:33 threatens the Enterprise of Things

In this Figure, we highlight a possible attack scenario where the main goal for the attacker is to disrupt the functioning of the sub-station, which can lead to a major blackout. ([Think of the 2015 attack on the Ukrainian grid.](#))

To accomplish their goal, the attacker can obtain [initial access](#) from the retail branch or from an employee's home office (points 1 in the figure), can [move laterally](#) to the enterprise HQ (point 2 in the figure), and from there they can finally reach the sub-station (point 3 in the figure) where they can cause the intended [impact](#): a [denial of view](#) and [control](#) that prevents operators from monitoring and controlling the physical processes.

Below, we provide the details on how an attacker can execute the steps above by exploiting some of the vulnerabilities in AMNESIA:33.

- Initial Access:** To obtain initial access from the retail branch, we assume the attacker manages to exploit one of the RCEs in AMNESIA:33, namely CVE-2020-25111. In the Technical Dive in Section 4.4, we discussed a proof of concept we run in our labs that shows how the vulnerability could be exploited. This vulnerability represents a good candidate for an attacker because it affects DNS, which is externally accessible and can usually traverse network boundaries, and because we have seen that several devices, including retail printers used for

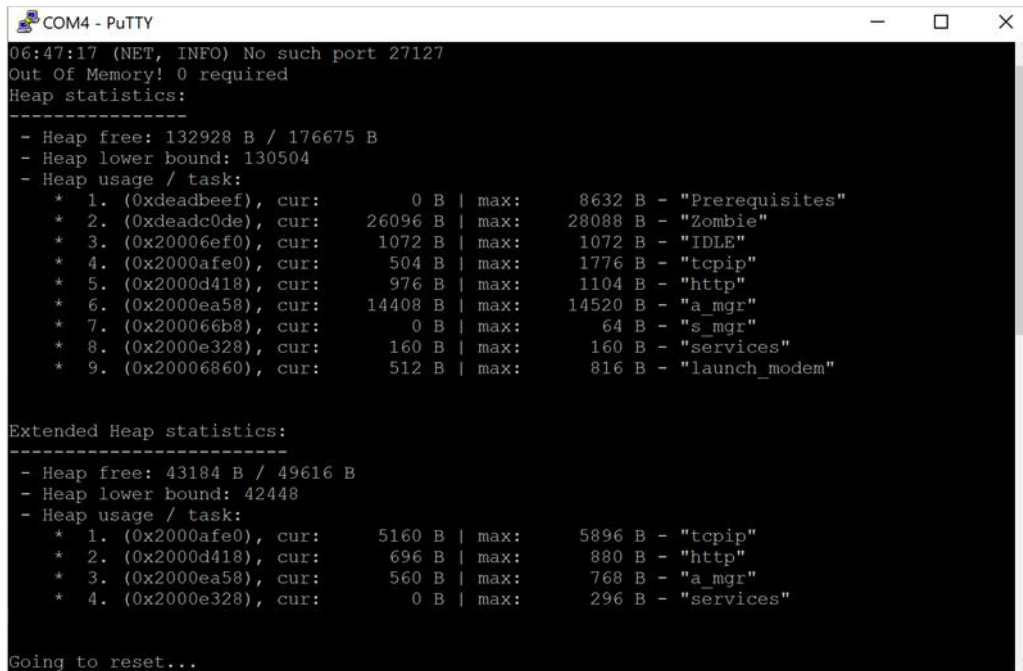
printing receipts, temperature monitors and building automation power meters run the vulnerable Nut/Net stack. Finally, these devices typically are connected to the enterprise network for, as two examples, remote maintenance or data transfers. In this scenario, **we assume that the attacker compromises a temperature monitor that is connected to a Building Automation Controller in the Enterprise HQ.** The compromise can happen by weaponizing the exploitation method discussed in the “Technical Dive In” (Section 4.4) with a payload such as a reverse shell, which would allow to gain a foothold into the target network. The caveat about CVE-2020-25111 is that it requires the attacker to be able to hijack DNS communications and reply to a legitimate request with a malicious packet. That hijacking can be done via a [man-in-the-middle](#) somewhere in the path between the request and the reply by exploiting a DNS server (either a local one in the target network or a more authoritative server) or by [registering no-longer used domains](#) in some cases. Another way for the attacker to gain initial access could be via an employee’s home office. The attacker **can target a vulnerable consumer IoT device** such as a home printer running the uIP stack with an RCE, such as the ICMP-based CVE-2020-25112. **This highlights the trend of consumer IoT devices representing more and more a threat for the extended enterprise.** A home printer may be connected to the HQ via VPN, which would allow an attacker to move from the employee’s home to the corporate HQ network.

- **Lateral Movement:** Once the attacker has obtained an initial foothold in the enterprise network, they can access vulnerable devices such as office printers or building automation controllers in the Enterprise HQ. In this scenario, we assume that the attacker will move to the building automation controller (which also runs Nut/Net, so the same CVE can be used) and [persist](#)

there to maintain their foothold. From that position, the attacker can now reach the devices in the sub-station.

- **Impact, Disrupting the Sub-station:** At the sub-station, the attacker has an ample choice of targets in the form of OT devices, such as RTUs running PicoTCP, protocol gateways running Nut/Net and industrial Ethernet switches running uIP. The attacker can first scan the network looking for the target. This can be done, for instance, by using active techniques such as the indirect information leak from mishandling TCP Urgent pointers that we discussed in the “Technical Dive In” about CVE-2020-17437 in Chapter 4. Since the goal is to disrupt network communications between the sub-station and the SCADA system, the attacker can directly DoS the RTU, which causes it to hang and reboot after a few seconds, thus interrupting the connection to the SCADA and the processing of input/output. Figure 21 shows the effect of exploiting CVE-2020-24337 on the device, as seen from a serial console connected to it. Notice that the device runs out of memory, dumps some internal information about the heap and resets. An interesting characteristic of this attack is that **it relies on a single malicious TCP packet to bring down the device**, which is very different from a DoS attack that relies on a rapid sequence of packets or a malicious command using an insecure OT protocol. This makes such an attack difficult to be detected out of the box by modern intrusion detection systems. Another possible attack is to DoS the industrial switch that connects the RTU to the network, thus disabling its communication in a different way, with the added impact of disabling the communication of other Ethernet-level devices. A third possibility is to DoS the protocol gateway, thus interrupting the processing of IOs read from serial devices, which may spread the effects of the attack to devices that are not even related to the RTU.

²For a discussion and demonstration of persistence on a building automation controller (unrelated to AMNESIA:33) see <https://www.forescout.com/securing-building-automation-systems-bas/>



```
COM4 - PuTTY
06:47:17 (NET, INFO) No such port 27127
Out Of Memory! 0 required
Heap statistics:
-----
- Heap free: 132928 B / 176675 B
- Heap lower bound: 130504
- Heap usage / task:
  * 1. (0xdeadbeef), cur: 0 B | max: 8632 B - "Prerequisites"
  * 2. (0xdead0de), cur: 26096 B | max: 28088 B - "Zombie"
  * 3. (0x20006ef0), cur: 1072 B | max: 1072 B - "IDLE"
  * 4. (0x2000afe0), cur: 504 B | max: 1776 B - "tcpip"
  * 5. (0x2000d418), cur: 976 B | max: 1104 B - "http"
  * 6. (0x2000ea58), cur: 14408 B | max: 14520 B - "a_mgr"
  * 7. (0x200066b8), cur: 0 B | max: 64 B - "s_mgr"
  * 8. (0x2000e328), cur: 160 B | max: 160 B - "services"
  * 9. (0x20006860), cur: 512 B | max: 816 B - "launch_modem"

Extended Heap statistics:
-----
- Heap free: 43184 B / 49616 B
- Heap lower bound: 42448
- Heap usage / task:
  * 1. (0x2000afe0), cur: 5160 B | max: 5896 B - "tcpip"
  * 2. (0x2000d418), cur: 696 B | max: 880 B - "http"
  * 3. (0x2000ea58), cur: 560 B | max: 768 B - "a_mgr"
  * 4. (0x2000e328), cur: 0 B | max: 296 B - "services"

Going to reset...
```

Figure 21 – Exploiting CVE-2020-24337 DoS on the RTU

6.1. Other possible attack scenarios

Other types of impact that could affect the enterprise itself, not just the sub-station, have to do with DoSing some devices found along the way.

For instance, we identified smoke alarms for smart homes and fire alarm control panels for smart buildings running the vulnerable stacks. These devices could be present either in the retail branch or the enterprise HQ and **disabling the communications on these systems or taking them offline by using any of the DoS vulnerabilities could open the way to physical attacks that aim to damage physical systems and ultimately even public safety.**

We also identified some vulnerable switches that are typically used in small offices or retail branches. DoSing these switches with a single malicious packet could cut off communications within the branch or between a branch and the enterprise HQ. **The impact of this on a retailer would be to cause massive delays and**

queues, which could be especially impactful during busy periods such as the holiday season. Similarly, a denial of service on the receipt printers also could cause delays in shopping.

Vulnerable temperature monitors are an [interesting target](#) because they are present not only on retailers, as shown in Figure 21, but also are critical to maintain the temperature in scenarios as diverse as the [food supply chain \(including processing, transportation and storage\)](#), healthcare and research facilities that store material that spoils at room temperature or data centers that store customer information and can [go offline in 15 minutes without appropriate cooling](#).

Exfiltrating enterprise data via compromised IoT devices is a scenario that is not attached to a single type of device since the device only needs to provide connectivity in this case, but has been shown [in practice before using other types of vulnerabilities](#) and could be replicated with an RCE from AMNESIA:33.

7. Effective IoT risk mitigation

When vulnerabilities in critical components, such as a TCP/IP stack, affecting millions of devices are discovered, it is important to know both which assets on a network are potentially affected and what kind of risk they are exposed to in order to prioritize patching and hardening efforts. **This kind of understanding requires a granular and context-aware visibility into network assets.**

When it comes to much of the IoT, however, this visibility is obscured by complex supply chains that propagate vulnerabilities. Because of the absence of a Software Bill of Materials, it is often very hard to determine whether a vulnerable software is in a device. Conversely, when vulnerabilities get discovered, it is very time-consuming to contact all potentially affected vendors. As a result, vulnerabilities that have been fixed in products in one industry might resurface to affect products in another [even a decade later](#) or issues found in one product, such as those affecting the [RTCS TCP/IP stack in the Smiths Medfusion 4000 infusion pump](#) might not travel fully up and down the supply chain, again leaving millions of devices unpatched and their owners unaware of the risk. Other examples of vulnerabilities affecting stacks that are reported by a single product include Siemens devices using [Nucleus NET](#) and [NicheStack](#). Examples of vulnerabilities affecting a product's TCP/IP stack where the exact stack is not mentioned include [Cisco IPSs](#) (2013), [Huawei switches](#) (2015), [Allen-Bradley safety devices](#) (2017), [Qualcomm modems](#) (2018) and several [Mitsubishi Electric devices](#) (2020). All those issues may still be present in millions of devices.

These challenges highlight the need for a large-scale study such as Project Memoria: Looking at specific devices is not enough to reveal the true state of IoT security and how to mitigate existing risks.

Merely being able to identify the operating system of a given asset, as many network visibility solutions

do, is not enough to address this issue. Consider that most TCP/IP stacks are modular components rather than tightly coupled to an RTOS. In some cases, such as VxWorks and IPnet or FreeRTOS and FreeRTOS+TCP, there is a designated 'default' stack, but there is nothing preventing system integrators from swapping it out for another one (and indeed we have seen example of devices running FreeRTOS and lwIP or picoTCP). With a few exceptions, blindly assuming that a device running a particular RTOS will run a particular TCP/IP stack might lead to inefficient risk management. For example, while the URGENT/11 advisory reported correctly that the IPnet stack could historically be found with ENEA's OSE, Green Hill's INTEGRITY and Microsoft's ThreadX, each of these use their own stacks since at least 2007 (respectively OSEnet, GHNet and NetX). Another example is Tesla's Gateway ECU, which runs [FreeRTOS](#) but does so with lwIP. One example that we found during AMNESIA:33 is a model in the MPL MAGBES family of industrial switches that runs components from three stacks: Most components are from the Nut/Net stack, but the SNMP and IGMP components come from other stacks (possibly [cycloneTCP](#) and the [Ralink/Mediatek SDK](#) based on a string analysis of the firmware).

Automated firmware analysis is also incapable of addressing this issue given the large number of embedded devices with closely guarded firmware, having encrypted or proprietary firmware formats and running exotic CPU architectures, while most automated analysis solutions tend to limit their support to Linux-based systems on ARM, MIPS or x86.

As such, IoT risk mitigation requires at the foundation a network-based granular, dynamic asset inventory capable of extracting information such as vendor, OS, firmware version and others via passive network fingerprints or active capabilities, while identifying TCP/IP stacks and their risk surface in a context-aware fashion in order to support enforcing mitigating controls and prioritized patch updates.

One lesson we learned while trying to identify devices running vulnerable stacks is that passive fingerprinting (for instance, based on [DHCP request fingerprinting](#)) is rarely enough. We often must resort to active capabilities that make use of quirks in the implementations of TCP/IP stacks to confirm if a device is vulnerable. These quirks include how stacks reply to ICMP requests and how they handle the TCP Urgent pointer, as we discussed in the Technical Dive In about CVE-2020-17437 in Chapter 4.

After identifying vulnerable devices, mitigations usually start with patching. But in the IoT/OT world, it is more and more common that patching is not an option because patches are not issued by vendors or cannot be applied to mission-critical systems. When this is the case, organizations should perform a thorough risk assessment of their networks to determine the required level of mitigation. The advantage of having a strong visibility foundation is that it gives network operators the confidence to isolate risky and critical devices that cannot be patched so that both their exposure to threats and the chance that they serve to propagate a threat or cause damage are minimized.

Below we identify some possible mitigating actions that asset owners and security operators can take to protect their networks from the TCP/IP vulnerabilities in AMNESIA:33 and also in other stacks:

- Disable or **block IPv6 traffic whenever it is not needed in the network** since several vulnerabilities in AMNESIA:33 and TCP/IP stacks in general are related to IPv6 components.
- Configure devices to **rely on internal DNS servers as much as possible** and closely monitor external DNS traffic since several vulnerabilities in AMNESIA:33 and TCP/IP stacks in general are related to DNS clients, which require a malicious DNS server to reply with malicious packets.
- **Monitor all network traffic for malformed packets** (for instance, having non-conforming field lengths or failing checksums) that try to exploit known vulnerabilities or possible 0-days since many vulnerabilities are related to IPv4 and other standard components of stacks. Anomalous and malformed IP traffic should be blocked, or its presence should be at least alerted to network operators.

8. Conclusion

In this first report of Project Memoria, we reported on AMNESIA:33, a set of 33 new vulnerabilities affecting four open source TCP/IP stacks and analyzed these findings in the context of previous similar vulnerabilities. Our main conclusions from this first report are as follows:

- TCP/IP stacks are vulnerable across the board. Despite having some examples of resilient stacks, such as lwIP, cycloneTCP and uC/TCP-IP, the rule of thumb is that at close inspection, they yield a large number of bugs.
- Many of these vulnerabilities arise from well-known bad software development practices, such as an absence of basic input validation and shotgun parsing.
- These patterns tend to generate bugs in all components, but their quantity and severity tend to correlate to increases in protocol complexity. Feature-rich protocols like DNS are particularly affected.
- The impact and exploitability of these vulnerabilities are highly device-specific, presenting challenges to adequate risk management.
- These vulnerabilities affect hundreds of vendors and millions of devices used currently in any enterprise, but complex IoT/OT supply chains make it challenging to determine which devices are affected and which are not. For the same reasons, these vulnerabilities are very hard to eradicate.

Based on the insights gained from this first study, **we plan on continuing investigating other stacks in detail** and specific vulnerability-prone components of stacks at a large scale. Other activities under Project Memoria may include larger discussions on the current process of vulnerability disclosure as it is applied to the emerging IoT world and specific recommendations to actively defend networks and organizations.

Don't just see it.
Secure it.™

Contact us today to actively
defend your Enterprise of Things.

[Download the white paper:](#) Discover how Forescout helps you actively defend against AMNESIA:33, including six best practices to protect your organization.

[View the webinar:](#) Listen to our experts describing the highlights of the research.

forescout.com/amnesia33/

research@forescout.com

toll free 1-866-377-8771



Forescout Technologies, Inc.
190 W Tasman Dr.
San Jose, CA 95134 USA

Toll-Free (U.S.) 1-866-377-8771
Tel (Intl) +1-408-213-3191
Support +1-708-237-6591

[Learn more at Forescout.com](https://forescout.com)

© 2020 Forescout Technologies, Inc. All rights reserved. Forescout Technologies, Inc. is a Delaware corporation. A list of our trademarks and patents can be found at <https://www.forescout.com/company/legal/intellectual-property-patents-trademarks>. Other brands, products or service names may be trademarks or service marks of their respective owners. Version 12_20