



# Automated 0-day discovery in 2021: Squashing the low-hanging fruit in widespread embedded software

**Daniel dos Santos | Shachar Menashe**

Forescout Research Labs | JFrog Security Research

TRACK 1

# About us

Daniel dos Santos, *Research Manager @ Forescout*

- Experience in security research, development and pentesting
- PhD in Computer Science, 30+ academic publications
- Speaker at Black Hat, x33fcon and others



Shachar Menashe, *Sr. Director Security Research @ JFrog*

- Experienced security researcher and architect
- BSc in Computer Science & Electrical Engineering
- Currently leading the security research teams @ JFrog



Co-authors:

- Stanislav Dashevskyi, Amine Amri, Jos Wetzels @ Forescout
- Asaf Karas, Denys Vozniuk @ JFrog

# Outline

## 1. Introduction

- Research Background
- INFRA:HALT
- Finding Vulnerabilities

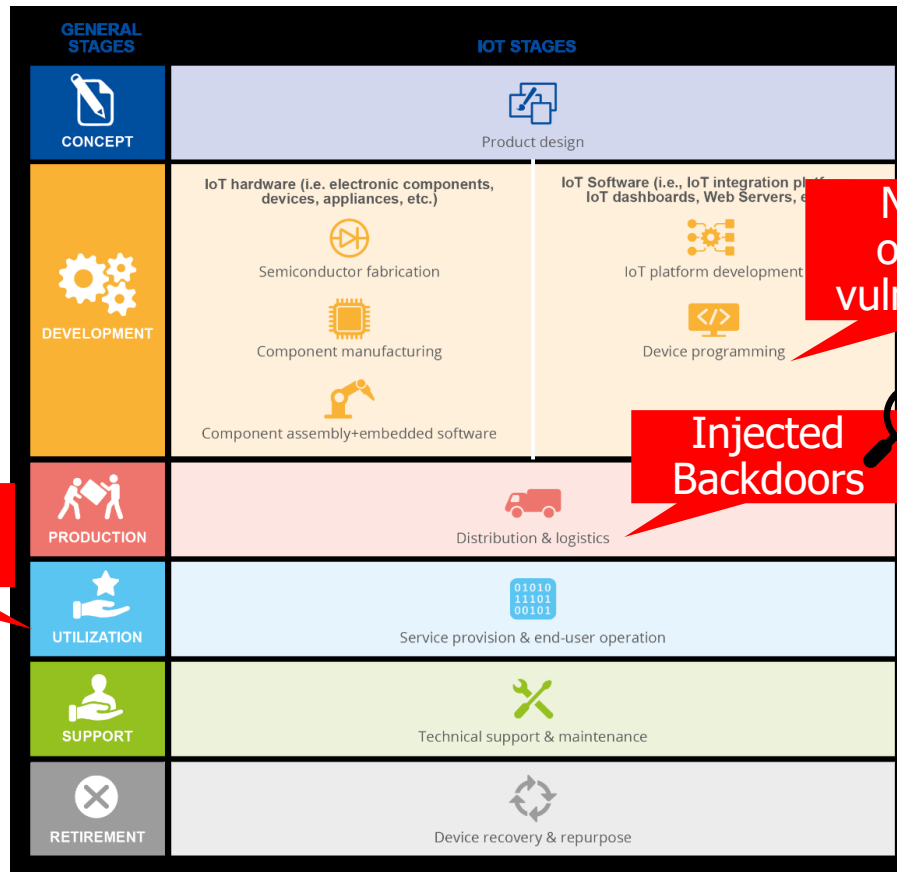
## 2. Automated Vulnerability Discovery

## 3. Mitigation

- Device vendors
- Network operators

## 4. Conclusion

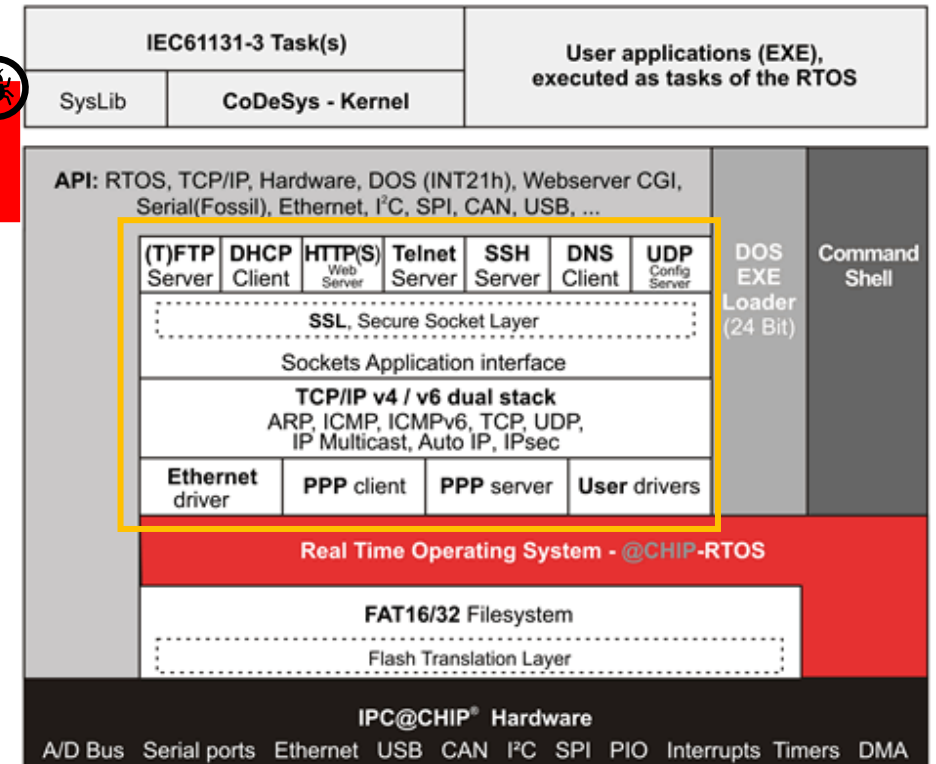
# Embedded Systems Supply Chain



Naturally occurring vulnerabilities

Injected Backdoors

Device Misconfiguration

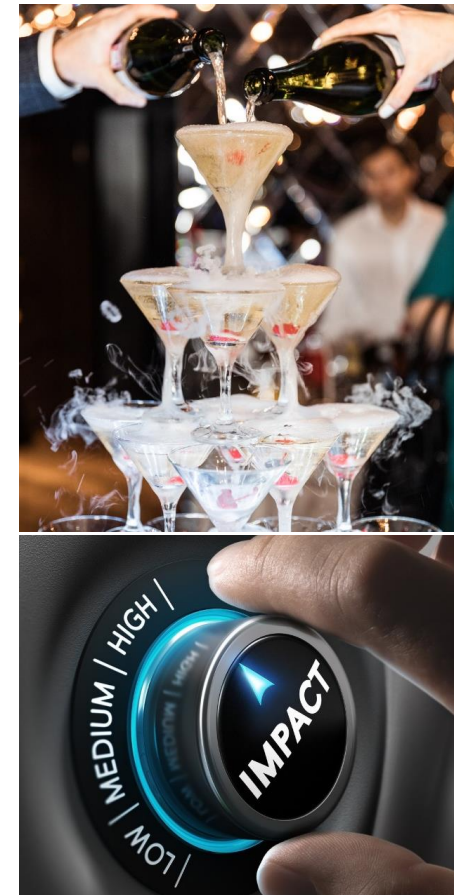


<https://www.enisa.europa.eu/publications/guidelines-for-securing-the-internet-of-things>

<http://smartbox.jinr.ru/doc/chip-rtos/software.htm>

# Why target protocol stacks

- Wide deployment – vulnerabilities **trickle down** the supply-chain to many vendors
- Absence of *Software Bill of Materials (SBOM)* - fixes in core stack **might never make it** to all OEM firmware
- Ancient code – good chance of finding exploitable bugs
- Externally exposed, often run as privileged, low-level component
- Patching issues + Long lifespans + Broad trickle-down = **High vulnerability lifespan = High attacker ROI**



# Previous work on TCP/IP stacks

Year	Research	Description
2019	<b>URGENT/11</b>	11 CVEs on VxWorks' IPnet
2020	<b>Ripple20</b>	19 CVEs on Treck TCP/IP
2020	<b>AMNESIA:33</b>	33 CVEs in 4 open-source stacks
2021	<b>NUMBER:JACK</b>	Predictable TCP ISN in 9 stacks (open and closed)
2021	<b>NAME:WRECK</b>	9 DNS client vulnerabilities in 4 stacks
2021	<b>INFRA:HALT</b>	14 CVEs on InterNiche stack

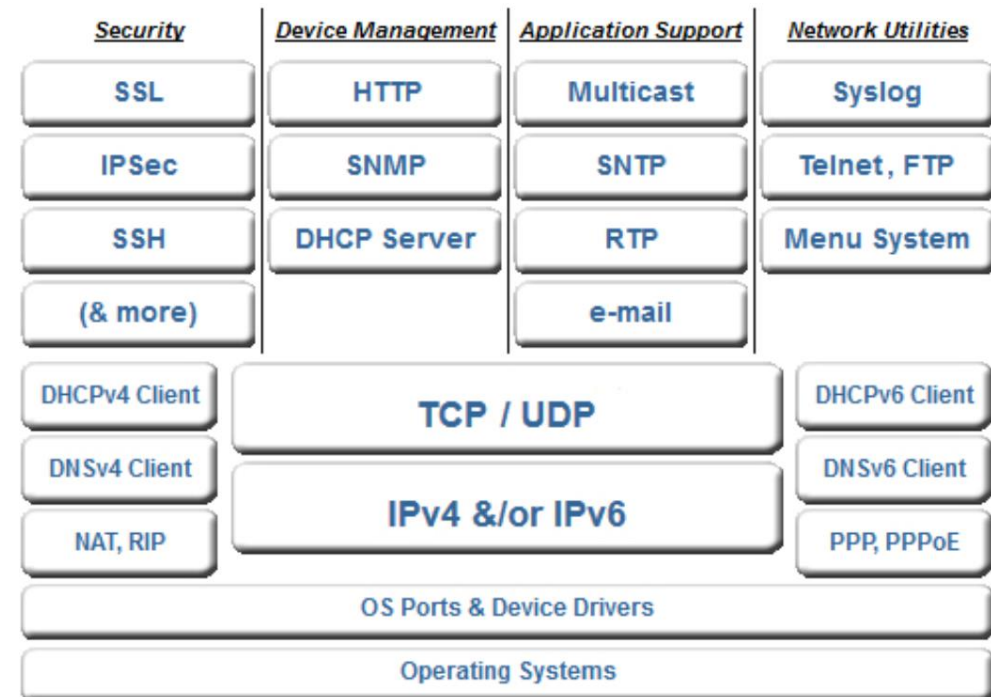
# INFRA:HALT

Vulnerabilities in the InterNiche  
embedded TCP/IP stack



# The target: What is NicheStack?

- Developed by InterNiche in the '90s, acquired by HCC Embedded in 2016
- Distributed in several flavors (IPv4, v6, dual, lite)
- Served as basis for other stacks (e.g., emNet)
- Popular in OT devices – previous research:
  - [Siemens](#): CVE-2019-9300, SegmentSmack variant affecting several devices
  - [Abbasi et al.](#): looking at S7 PLCs, found and compiled stack source-code leaked via OEM



[https://ww1.microchip.com/downloads/en/Site\\_Resource/NicheStack%20IPv4-ProductBrief.pdf](https://ww1.microchip.com/downloads/en/Site_Resource/NicheStack%20IPv4-ProductBrief.pdf)

# The vulnerabilities found

- 14 CVEs
- 5 components affected
  - DNS client
  - HTTP server
  - ICMP
  - TCP
  - TFTP server
- 2 RCEs
  - CVE-2020-25928 (DNS)
  - CVE-2021-31226 (HTTP)
- Found *manually* and/or *automatically*

#	CVE	Impact	CVSS	Component Affected
1	CVE-2020-25928	Remote Code Execution	9.8	DNS client
2	CVE-2021-31226	Remote Code Execution	9.1	HTTP server
3	CVE-2020-25767	Denial of Service	7.5	DNS client
4	CVE-2020-25927	Denial of Service	7.5	DNS client
5	CVE-2021-31227	Denial of Service	7.5	HTTP server
6	CVE-2021-27565	Denial of Service	7.5	HTTP server
7	CVE-2020-35683	Denial of Service	7.5	ICMP
8	CVE-2020-35684	Denial of Service	7.5	TCP
9	CVE-2021-31400	Denial of Service	7.5	TCP
10	CVE-2021-31401	Denial of Service	7.5	TCP
11	CVE-2021-36762	Denial of Service	7.5	TFTP server
12	CVE-2020-35685	TCP spoofing	7.5	TCP
13	CVE-2020-25926	DNS cache poisoning	4	DNS client
14	CVE-2021-31228	DNS cache poisoning	4	DNS client

# RCE1: CVE-2020-25928 (DNS)

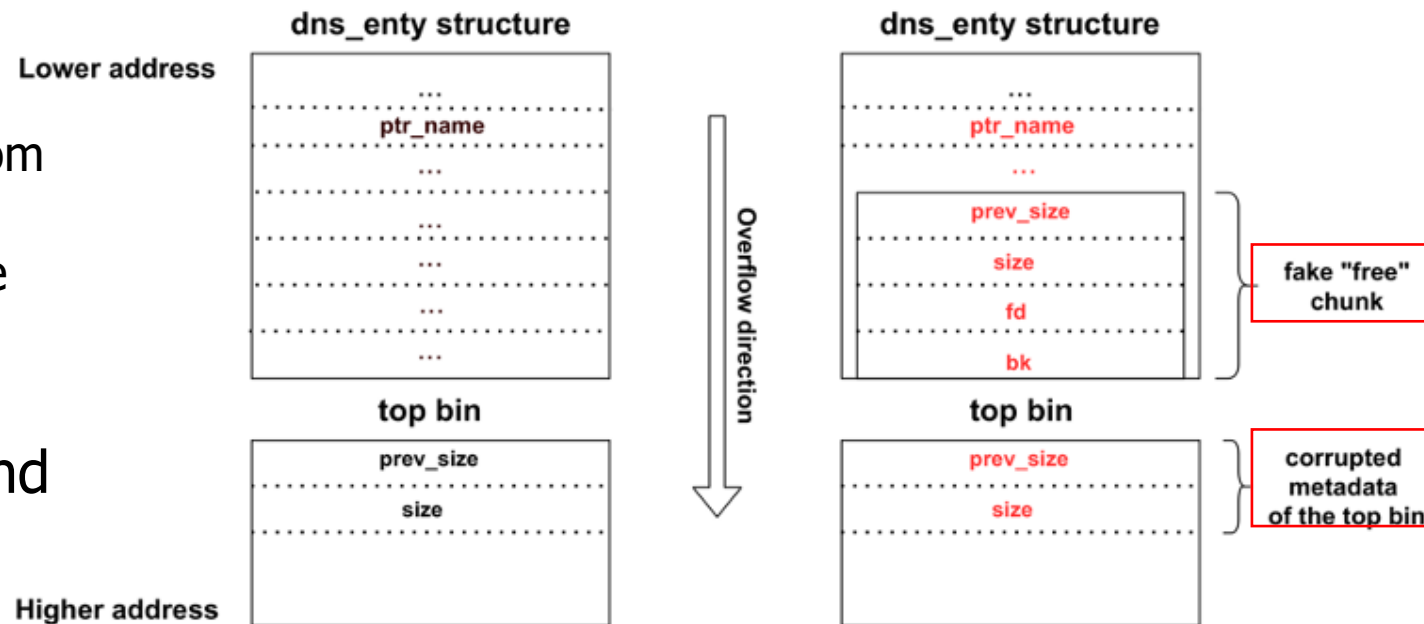
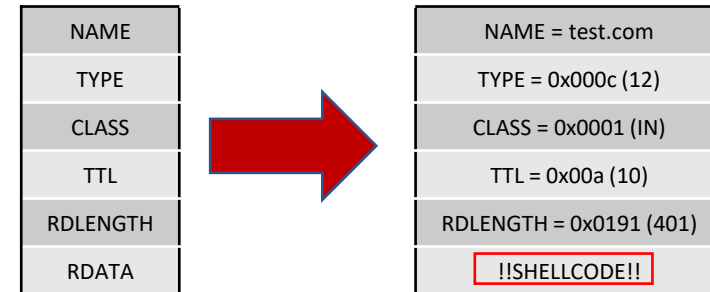
- Found manually, based on anti-pattern from NAME:WRECK
  - Similar to CVE-2020-27009 on Nucleus NET
- Resource Record length (RDLENGTH) of DNS responses is not checked
  - Attackers can specify arbitrary RDLENGTH and overflow next field (RDATA)
  - A buffer for RDATA is allocated on the heap
  - There are usually no exploit mitigations

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

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

# Exploiting CVE-2020-25928

- Achieved RCE with the classical "unlink" technique
  - No safe unlinking
- Easy for attackers to spoof DNS records
  - Source port and TXID aren't random
    - CVE-2020-25926, CVE-2021-31228
  - responses from any IP address are accepted
- Shellcode uses the stack API to perform TCP handshake and send further malicious packets



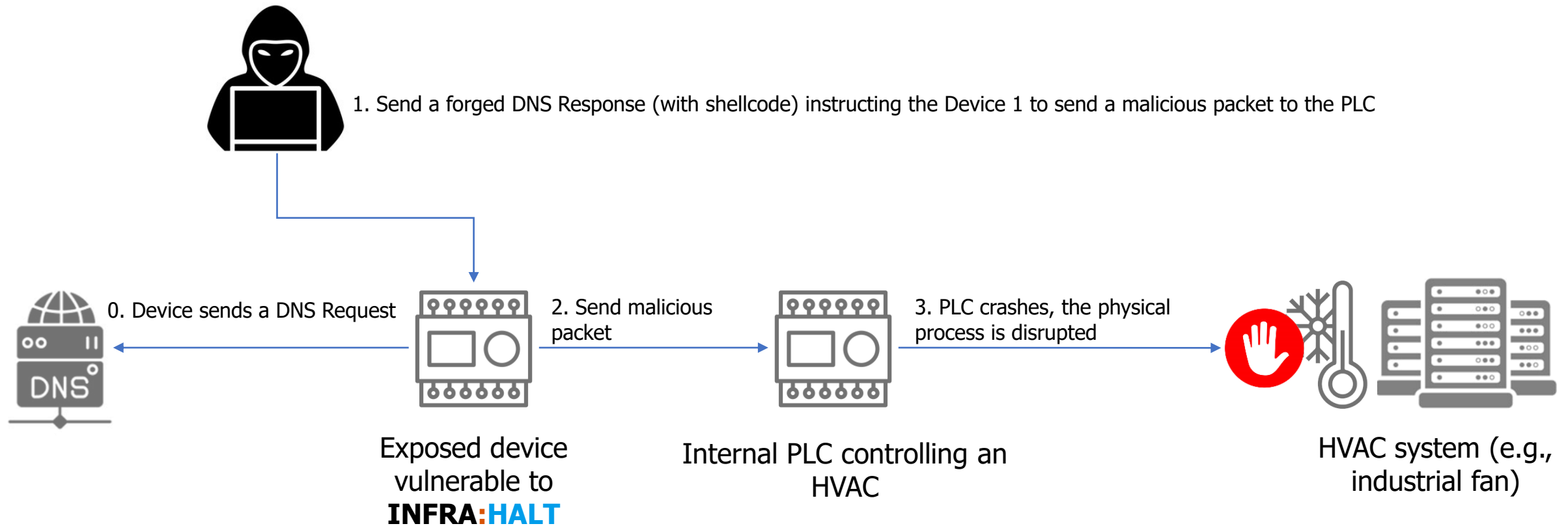
# RCE2: CVE-2021-31226 (HTTP)

- Found automatically – *more details later*
- Occurs when parsing the HTTP POST Request URI field: <http://example.org/path/to/file?param42>
- A request string of more than 52 bytes may cause a (heap) buffer overflow
- RCE can be achieved similar to CVE-2020-25928 (more careful heap shaping required)

```

v Hypertext Transfer Protocol
  v POST /AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAzzzz HTTP/1.1\r\n
    > [Expert Info (Chat/Sequence): POST /AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAzzzz HTTP/1.1\r\n]
      Request Method: POST
      Request URI: /AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAzzzz
      Request Version: HTTP/1.1
      Host: 192.168.100.2\r\n
      Content-Type: multipart/form-data;boundary=xxxxxx\r\n
  v Content-Length: 0\r\n
    [Content length: 0]
    Connection: close\r\n
    \r\n
    [Full request URI: http://192.168.100.2/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAzzzz]
    [HTTP request 1/1]
  
```

# Attack scenario



<https://www.youtube.com/watch?v=plgtt1BD-nI>

# Supply Chain Impact

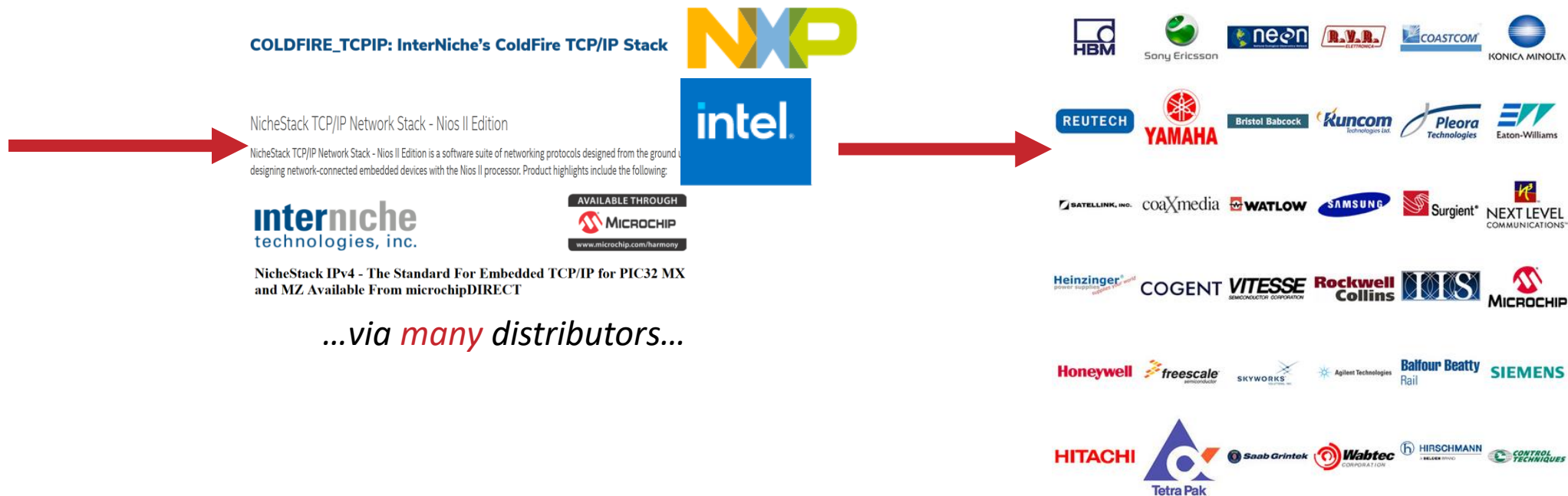
## NicheStack Products

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

Many offerings

<https://web.archive.org/web/20170213060851/http://www.iniche.com:80/company/manylogos.php>

<https://t.me/learnignets> Automated 0-day discovery in 2021 | Daniel dos Santos & Shachar Menashe



**interniche**  
Networking Protocol Software for Embedded Processors  
Click Here to Contact Us via Email.  
Also, consider using our Contact Form

- Networking Stacks: TCP/IP, IPv6, Bluetooth, PPP
- Embedded Security: SSL, IPsec, etc.
- Apps & Device Management: HTTP, SNMP, DHCPv6 & more
- Device Support: File Systems, RTOS, USB, Ports
- Demos & Other Formats: Pre-Compiled Licos, Free Demos

Just Some Customers of InterNiche's Networking Source Code

...end up in many ways in many products from many vendors

# Finding the vulnerabilities

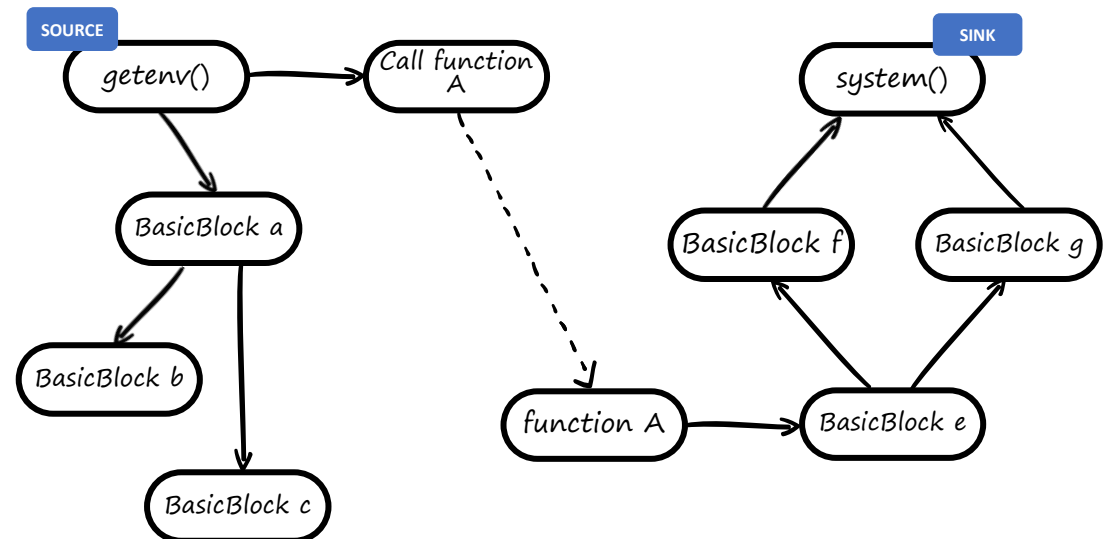
- Input
  - Leaked source code of v3.1 (as mentioned in previous research)
  - Binary demo of more recent version (previously available on vendor's website)
- Manual analysis based on known anti-patterns
  - AMNESIA:33 – integer overflows, lack of bounds checks
  - NUMBER:JACK – weak ISN
  - NAME:WRECK – DNS compression and several others – see <https://github.com/Forescout/namewreck>
- The stack matched almost all the known anti-patterns
  - Didn't analyze IPv6 – not available in the source
  - Great result, but ***lots of work and potentially missed issues...***
- **...enter automated Vulnerability Discovery**

# Automated Vulnerability Discovery

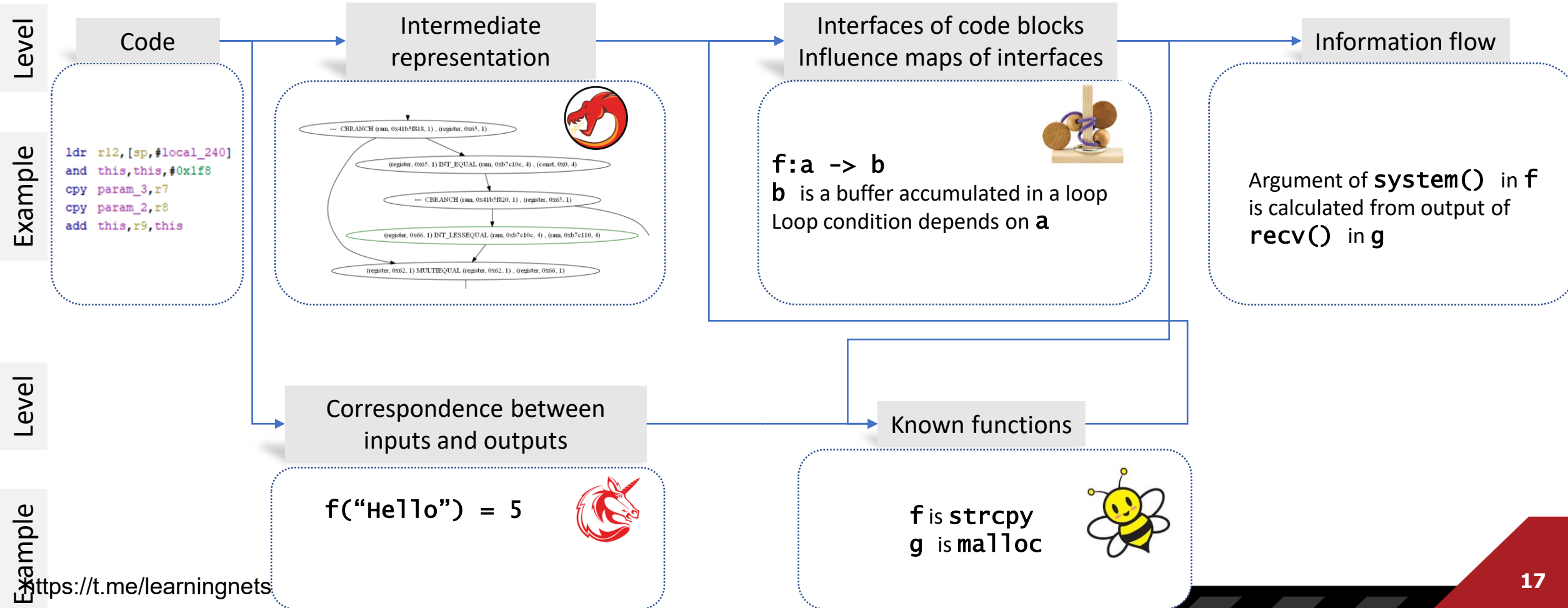
Overcoming the limitations of manual research

# Method of operation – high level

- #1 – Map possible user input **sources**
- #2 – Map possible dangerous **sinks**
- #3 – Find **unfiltered** data flow between #1 & #2



# Method of operation - detailed



# Higher level analysis – Ghidra P-Code

- Ghidra's decompiler lifts ASM to high-level IR
- This allows for easier, cross-architecture analysis

```
memset(&rcvdq,0,0x14);
void *to_netmain = NULL;
void *to_nettick = NULL;
cli_install_menu(&netmain_nt);
```

*Original source*

```
mov.w    r1,#0x0
mov.w    r2,#0x14
bl       memset
movw     r3,#0x3a38
movt     r3,#0x2000
mov.w    r2,#0x0
tr       r2,[r3,#0x0 ]=>to_netmain
movw     r3,#0x3a3c
movt     r3,#0x2000
mov.w    r2,#0x0
tr       r2,[r3,#0x0 ]=>to_nettick
movw     r0,#0xc4c
movt     r0=>netmain_nt ,#0x2000
bl       cli_install_menu
```

*Disassembly*

OUTPUT                      P-CODE                      INPUT 1                      INPUT 2

```
(unique, 0x1000004c, 4) PTRSUB (const, 0x0, 4) , (const, 0x20003f0c, 4) // &rcvdq
--- CALL (ram, memset, 8) , (unique, 0x1000004c, 4) , (const, 0x0, 4) , (const, 0x14, 4)
(ram, 0x20003a38, 4) COPY (const, 0x0, 4) // *to_netmain = NULL
(ram, 0x20003a3c, 4) COPY (const, 0x0, 4) // *to_nettick = NULL
(unique, 0x10000030, 4) PTRSUB (const, 0x0, 4) , (const, 0x20000c4c, 4) // &netmain_nt
--- CALL (ram, cli_install_menu, 8) , (unique, 0x10000030, 4)
```

*Decompilation*

# Mapping user input sources

- Usually harder than mapping sinks
- High-accuracy sources
  - Syscalls – *recv* (network), *getenv* (local), *fread* (local)
  - Reading from well-known input MMIO (ex. UART, BLE)
    - See [Ghidra SVD-Loader](#)
- Score-based sources
  - *ntohs* / *ntohl* (assumes this converts network integers)
    - Note these can often be inlined!
  - Functions that reference well-known protocol strings
    - In this case, return value and all arguments and will be treated as sources

# Example of protocol excerpts

## HTTP

```
pcVar6 = strstr(pcp,"Content-Length:");
if (pcVar6 == (char *)0x0) {
    ht_senderr(hp,400,"Content-length required");
    hp->state = 8;
    return;
}
lVar7 = atoi(pcVar6 + 0xf);
hp->contentlen = lVar7;
if ((hp->rsize - (int)(pcVar4 + -(int)pcVar9) < hp-
>contentlen) &&
    (pcVar5 = strstr(pcVar5,"multipart/form-
data"), pcVar5 == (char *)0x0)) {
```

## FTP

```
v10 = command(L"USER %s");
v11 = GetProcessHeap();
...
v10 = command(L"PASS %s");
v14 = GetProcessHeap();
...
v10 = command(L"ACCT %s");
```

- More encountered strings = higher confidence score

# Mapping sinks

- Basic sinks
  - Command injection – *system* / *popen* etc.
  - Buffer overflow – *memcpy* / *strcpy* etc.
    - In memcpy case, check that both source and length are user-controlled
- Advanced sinks
  - Inline copy operations / copy loops
  - Integer overflow leading to buffer overflow

```
while ( 1 )
{
    cur = src + 1;
    if ( cur2 == '@' )
        break;
    ++src;
    dst[v3] = cur2;
    cur2 = (unsigned __int8)*cur;
    ++v3;
    if ( !*cur )
        goto exit_loop;
}
```

# libc detection via emulation

- libc might be statically linked (ex. RTOS binary blob)
- This means – no function symbols!
- But we need function names such as “strcpy” for our sinks
- Our solution – function divination via emulation
- Inputs
  - Expected function prototype
  - Set of matching inputs & outputs
- Outputs
  - All functions with matching behavior

```
class TestStrcat(Function):  
    NAME = "strcpy"  
    PARAM_DEREFS = [1, 1]  
  
    STRING = b"Hello,"  
    STRING2 = b" world !"  
  
    def probe(self):  
        ptr = self.alloc_mem(len(self.STRING) + len(self.STRING2) + 1, write=True)  
        self.write_string(ptr, self.STRING)  
        ptr2 = self.alloc_string(self.STRING2)  
  
        result = self.call(ptr, ptr2)  
  
        return (  
            result == ptr  
            and self.memcmp(ptr, self.STRING + self.STRING2)  
            and self.memcmp(ptr2, self.STRING2)  
        )  
  
    PROBES = Probe(probe)
```

# Data flow analysis – Ghidra's API

- Ghidra provides basic **intra-function** DFA
- ex. [getForwardSlice / getForwardSliceToPCodeOps](#)

```
phVar2 = hp->upload;
iVar3 = phVar2->curlen;
n = phVar2->boundarylen;
if (iVar3 + inlen < (int)n) {
    memcpy(phVar2->pbuf + iVar3, start, inlen);
    iVar1 = strcmp(phVar2->pbuf, phVar2->boundary, iVar3 + inlen);
    if (iVar1 == 0) {
        phVar2->curlen = iVar3 + inlen;
        return (char *)0x0;
    }
}
```

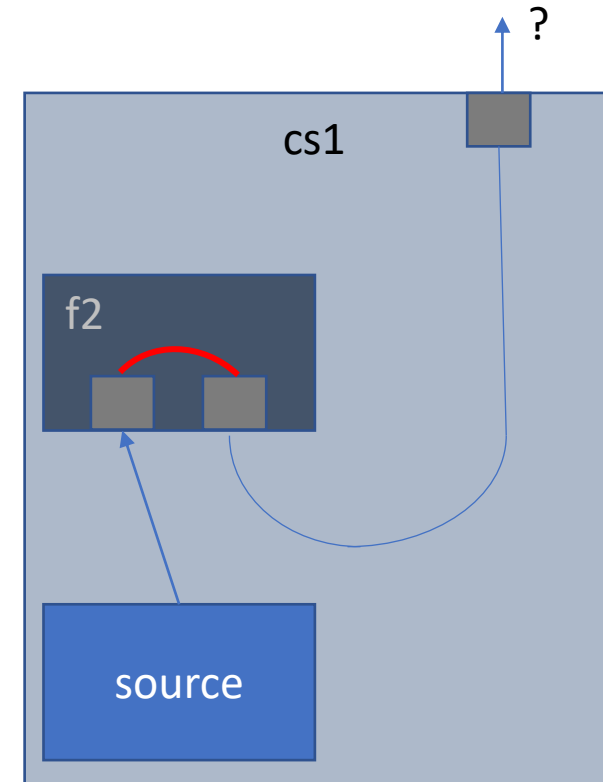
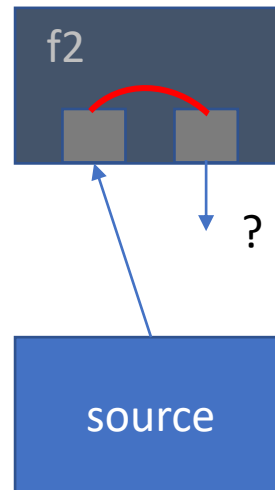
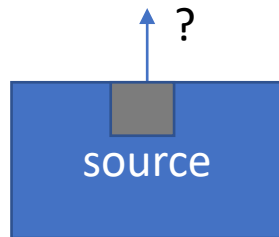
- Doesn't handle stack variables
- Does not propagate outside of function or into child functions

# Data flow analysis – expanding on Ghidra

- Do any of the defined source variables “reach” a sink function?

```
int cs1(int* a) {
  int d;
  source(&d);
  f2(&d, a);
}
```

```
int f2(int* a, int* b) {
  *b = *a;
}
```



# Data flow analysis – basic filtering

- Specific operations in the data flow path can make the vulnerability unexploitable
- Classic examples of these can be statically detected
  - Buffer overflow – size checks
  - Command injection – shell metacharacter filtering
  - General – data anchoring

```
char buf[50];
if (strlen(userinput) >= 50) {
    return ERR;
}
strcpy(buf, userinput);
```

```
char *res = strpbrk(userinput, "*;`$}|&<>");
if (NULL != res) {
    // Shell metacharacter detected!
    return ERR;
}
```

```
char buf[50];
if (strcmp(userinput, "fixed_input")) {
    return ERR;
}
// userinput == "fixed_input"
strcpy(buf, userinput);
```

# Advanced filtering with symbolic execution

- Filtering may be too exotic to detect via a fixed list of static cases

```
unsigned int a = read_int();
unsigned int b = read_int();
if( a > 1 ){
    if( b < 20) {
        if( a * b > 30){
            func_one();
        } else {
            func_two();
        }
    } else {
        func_three();
    }
} else {
    func_four();
}
```

$X > 1$  and  $Y < 20$

$X > 1$  and  $Y < 20$   
and  
 $X * Y > 30$

$X > 1$  and  $Y < 20$   
and  
 $X * Y < 30$

# Advanced filtering with symbolic execution

- Very compute intensive, must be restricted to pre-observed code blocks from static analysis

```
char *userinput = getenv("UNSAFE_VAR"); // No constraints on userinput
... // A lot of processing on userinput
system(userinput); // Many constraints on userinput
```

- Last line might have many constraints
  - `strlen(userinput) == 3 && userinput[1] = 'A'` etc.
- Add a custom constraint and check for satisfiability
  - `userinput[i] = ``' for i in strlen(userinput)`

# Detecting CVE-2021-31228

- HTTP server DoS
- Signed comparison leading to (huge) overflow

```
char * getbndsrch(htupload *htup,char *cp,int len,int *err)
{
    if (len < htup->boundarylen) { // Signed comparison!
        memcpy(htup->pbuf,cp,len);
        ...
    }
}
```

# Detecting CVE-2021-31228 (2)

- Source detection – through dynamic stdlib mapping (atol)
- Source function “ht\_readmsg” also flagged due to HTTP strings

```
void ht_readmsg(httpd *hp)
{
    ...
    pcVar6 = strstr(pcp,"Content-Length:");
    ...
    lVar7 = unknown_func(pcVar6 + 0xf); // Actually "atol". Parse "Content-Length"
    hp->source_field = lVar7; // Mark the field as having user input
    if ((hp->rxsize - (int)(pcVar4 + -(int)pcVar9) < hp->contentlen) &&
        (pcVar5 = strstr(pcVar5,"multipart/form-data"), pcVar5 == (char *)0x0)) {
        ...
    }
}
```

# Advantage of dynamic divination

```
long unknown_func(_reent *rptr, char *nptr, char **endptr, int base)
{
    pbVar5 = (byte *)nptr;
    do {
        pbVar2 = pbVar5;
        pbVar5 = pbVar2 + 1;
        uVar7 = (uint)*pbVar2;
        bVar9 = __ctype_ptr__[uVar7 + 1] & 8;
    } while (bVar9 != 0);
    if (uVar7 == 0x2d) {
        uVar7 = (uint)*pbVar5;
        bVar9 = 1;
        pbVar6 = pbVar2 + 2;
    }
    else {
        pbVar6 = pbVar5;
        if (uVar7 == 0x2b) {
            pbVar6 = pbVar2 + 2;
            uVar7 = (uint)*pbVar5;
        }
    }
    if (base == 0) {
        if (uVar7 != 0x30) {
            base = 10;

```

# Detecting CVE-2021-31228 (3)

- “hp” struct and fields tracked via DFA through multiple functions



# Detecting CVE-2021-31228 (4)

- Eventually a memcpy sink is reached (there are two)
- Vulnerability classified as signed comparison
  - Without "if" check -> classified as heap overflow
  - With unsigned "if" check -> classified as non-vuln

```
char * getbndsrch(htupload *htup, char *cp, int len, int *err)
{
    if (len < htup->boundarylen) { // Signed comparison
        memcpy(htup->pbuf, cp, len); // Buffer overflow sink
        ...
    }
}
```

# Mitigation

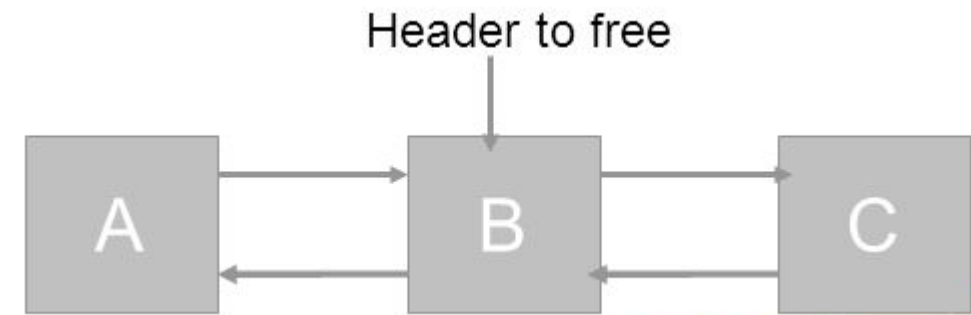
What can we do about these widespread vulnerabilities?

# For device vendors

## 1. Enable common vulnerability mitigations

- **Safe unlinking**
- Stack canaries
- ASLR
- FORTIFY\_SOURCE

B->Flink->Blink == B->Blink->Flink == B



## 2. Employ SAST and DAST scanning solutions

- As shown, some issues can be found automatically

# For network operators

## 1. Know what is on your network

- Assess risk upon connect and continuously
- Patch devices if possible

## 2. Segment to mitigate risk

## 3. Monitor the network for malicious packets

- Vulnerabilities in TCP/IP stacks tend to be very similar

Affected component	Mitigation Recommendation
DNS client	<ul style="list-style-type: none"> <li>• Disable the DNS client of the device if possible and not needed</li> <li>• Block DNS traffic if not needed</li> <li>• Using internal DNS servers is not sufficient because there are several vulnerabilities that facilitate DNS spoofing attacks</li> </ul>
HTTP / TFTP	<ul style="list-style-type: none"> <li>• Disable HTTP / TFTP server of the device if not needed</li> <li>• Whitelist connections and block others</li> </ul>
TCP	<ul style="list-style-type: none"> <li>• Monitor traffic for malformed IPv4/TCP packets</li> <li>• Drop these malformed packets at firewalls</li> </ul>
ICMP	<ul style="list-style-type: none"> <li>• Monitor traffic for malformed ICMPv4 packets</li> <li>• Drop these malformed packets at firewalls</li> </ul>

# Conclusion

# Discussion

- Vulnerability finding today: manual + automated
  - Automated can find low-hanging fruits much easier and faster
  - Manual still useful for more complex issues
- More vulnerabilities means more *vulnerabilities to disclose*
  - INFRA:HALT took 9 months
  - Currently, there is very limited involvement of asset owners
- Identifying vulnerable devices without firmware analysis is challenging
  - Lack of SBOM, opaque documentation, few network banners, etc.
  - Some vendors still investigating impact of AMNESIA:33 almost a year after initial disclosure

# Key takeaways

- TCP/IP stacks have critical vulnerabilities that trickle down the supply chain
  - Other popular software components could have similar impact
- Automated vulnerability discovery helps in identifying many of those at a large-scale
  - Turning point for the community, soon even more vulnerabilities will be found even faster
- Mitigation of these widespread issues involves both device vendors and network operators
  - Since they affect legacy but active devices, just waiting for patches is not a good solution
- Learn more at
  - [Forescout's Project Memoria](#)
  - [JFrog's blog](#)



# Thank You for Joining Us

Join our Discord channel to discuss more or ask questions

<https://discord.gg/dXE8ZMvU9J>