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
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

Device Misconfiguration

Naturally occurring vulnerabilities

Injected Backdoors


Automated 0-day discovery in 2021 | Daniel dos Santos & Shachar Menashe

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

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

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*

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

NAME = test.com
TYPE = 0x000c (12)
CLASS = 0x0001 (IN)
TTL = 0x000a (10)
RDLENGTH = 0x0191 (401)
!!SHELLCODE!!
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)
1. **Device sends a DNS Request**

2. **Send malicious packet**

3. **PLC crashes, the physical process is disrupted**

Exposed device vulnerable to **INFRA:HALT**

Internal PLC controlling an HVAC

HVAC system (e.g., industrial fan)

https://www.youtube.com/watch?v=plgtt1BD-nI
Supply Chain Impact

NicheStack Products

Many offerings

...via many distributors...

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


Automated 0-day discovery in 2021 | Daniel dos Santos & Shachar Menashe
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

Automated 0-day discovery in 2021 | Daniel dos Santos & Shachar Menashe
Method of operation - detailed

**Code**

```plaintext
ldr r10,[sp,#local_240]
and this,thi,#$null
cpy param_3,c7
cpy param_3,c8
add this,c9,this
```

**Intermediate representation**

- Argument of `system()` in `f` is calculated from output of `recv()` in `g`
- `f` is a buffer accumulated in a loop depending on `a`

**Interfaces of code blocks**

**Influence maps of interfaces**

**Information flow**

- `f` is `strcpy`
- `g` is `malloc`

**Correspondence between inputs and outputs**

- Example: `f("Hello") = 5`
Higher level analysis — Ghidra P-Code

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

```
memset(&rcvdq, 0x014);
void *to_netmain = NULL;
void *to_nettick = NULL;
cli_install_menu(&netmain_nt);
```

Original source

```
(void *) to_netmain = NULL;
(void *) to_nettick = NULL;
cli_install_menu(&netmain_nt);
```

Decomposition

```
(void, 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, 0x20003c4c, 4) // &netmain_nt
-- CALL (ram, cli_install_menu, 8), (unique, 0x10000030, 4)
```

Disassembly

```
  mov.w  r1, #0x00
  mov.w  r2, #0x14
  bl     memset

  movw   r3, #0x3a38
  movt   r3, #0x2000
  mov    r2, #0x00
  tr     r2, [r3, #0x0] => to_netmain

  movw   r3, #0x3a3c
  movt   r3, #0x2000
  mov    r2, #0x00
  tr     r2, [r3, #0x0] => to_nettick

  movw   r0, #0xc4c
  movt   r0 => netmain_nt, #0x2000
  bl     cli_install_menu
```
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

cVar6 = stristr(pcp,"Content-Length:");
if (cVar6 == (char *)0x0) {
    ht_senderr(hp,400,"Content-length required");
    hp->state = 8;
    return;
}
 iVar7 = atol(cVar6+0xf);
hp->contentlen = iVar7;
if ((hp->rxsize - (int)(pcVar4 + -(int)pcVar9) < hp->contentlen) &&
    (cVar5 = stristr(cVar5,"multipart/form-data"), cVar5 == (char *)0x0))
{

FTP

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

```c
while ( 1 )
{
    cur = src + 1;
    if ( cur2 == '\0' )
        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
Data flow analysis – Ghidra’s API

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

```c
phVar2 = hp->upload;
iVar1 = phVar2->curlen;
n = phVar2->boundarylen;
if (iVar1 + inlen < (int)n) {
    memcpy(phVar2->pbuf + iVar1,start,inlen);
iVar1 = strncmp(phVar2->pbuf,phVar2->boundary,iVar1 + inlen);
    if (iVar1 == 0) {
        phVar2->curlen = iVar1 + 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?

```c
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

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

```c
char *res = strpbrk(userinput, "*;${}\&<>");
if (NULL != res) {
    // Shell metacharacter detected!
    return ERR;
}
strcpy(buf, userinput);
```
Advanced filtering with symbolic execution

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

```c
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

```c
void ht_readmsg(httpd *hp)
{
    ...
    pcVar6 = stristr(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 = stristr(pcVar5,"multipart/form-data"), pcVar5 == (char *)0x0)) {
        ...
    }
```
Advantage of dynamic divination

```c
long unknown_func(_reent *rptr, char *nptr, char **endptr, int base)
{
    byte *pbVar5 = nptr;
    do {
        byte *pbVar2 = pbVar5;
        byte *pbVar5 = pbVar2 + 1;
        uint *uVar7 = pbVar2;
        byte bVar9 = __ctype_ptr__[uVar7 + 1] & 8;
    } while (bVar9 != 0);
    if (uVar7 == 0x2d) {
        uint *uVar7 = pbVar5;
        byte bVar9 = 1;
        byte *pbVar6 = pbVar2 + 2;
    } else {
        byte *pbVar6 = pbVar5;
        if (uVar7 == 0x2b) {
            byte *pbVar6 = pbVar2 + 2;
            uint *uVar7 = pbVar5;
        }
    }
    if (base == 0) {
        if (uVar7 != 0x30) { // Assuming uint is byte here
            base = 10;
        }
    }
}```
Detecting CVE-2021-31228 (3)

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

(Data Reference)
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

```c
char * getbndsrc(char *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

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

<table>
<thead>
<tr>
<th>Affected component</th>
<th>Mitigation Recommendation</th>
</tr>
</thead>
</table>
| DNS client         | • Disable the DNS client of the device if possible and not needed  
|                    | • Block DNS traffic if not needed  
|                    | • Using internal DNS servers is not sufficient because there are several vulnerabilities that facilitate DNS spoofing attacks |
| HTTP / TFTP        | • Disable HTTP / TFTP server of the device if not needed  
|                    | • Whitelist connections and block others |
| TCP                | • Monitor traffic for malformed IPv4/TCP packets  
|                    | • Drop these malformed packets at firewalls |
| ICMP               | • Monitor traffic for malformed ICMPv4 packets  
|                    | • Drop these malformed packets at firewalls |
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