July 24, 2019

U-Boot NFS RCE Vulnerabilities (CVE-2019-14192)

 Fermin J. Serna

This post is about 13 remote-code-execution vulnerabilities in the U-Boot boot loader, which I found with my colleagues Pavel Avgustinov and Kevin Backhouse. The vulnerabilities can be triggered when U-Boot is configured to use the network for fetching the next stage boot resources.

Please note that the vulnerability is not yet patched at https://gitlab.denx.de/u-boot/u-boot, and that I am making these vulnerabilities public at the request of U-Boot's master custodian Tom Rini. For more information, check the timeline below.

MITRE has issued the following CVEs for the 13 vulnerabilities: CVE-2019-14192, CVE-2019-14193, CVE-2019-14194, CVE-2019-14195, CVE-2019-14196, CVE-2019-14197, CVE-2019-14198, CVE-2019-14199, CVE-2019-14200, CVE-2019-14201, CVE-2019-14202, CVE-2019-14203 and CVE-2019-14204

What is U-Boot?

Das U-Boot (commonly known as "the universal boot loader") is a popular primary bootloader widely used in embedded devices to fetch data from different sources and run the next stage code, commonly (but not limited to) a Linux Kernel. It is commonly used by IoT, Kindle, and ARM ChromeOS devices.

U-Boot supports fetching the next stage code from different file partition formats (ext4 as an example), but also from the network (TFTP and NFS). Please note, U-boot supports verified boot, where the image fetched is checked for tampering. This mitigates the risks of using insecure cleartext protocols such as TFTP and NFS. so any vulnerability before the signature check could mean a device jailbreak.

I am using U-boot, am I affected?

These vulnerabilities affect a very specific U-Boot configuration, where U-Boot is instructed to use networking. Some of these vulnerabilities exist in the NFS parsing code but some others exist in the generic TCP/IP stack.

This configuration is commonly used on diskless IoT deployment and during rapid development.

What is the impact?

Through these vulnerabilities an attacker in the same network (or controlling a malicious NFS server) could gain code execution at the U-Boot powered device. Due to the nature of this vulnerability, exploitation does not seem extremely complicated, although it could be made more challenging by using stack cookies, ASLR or other memory protection runtime and compile time mitigations.

Understood, what are the vulnerabilities?

The first vulnerability was found in 2 very similar occurrences via source code review and we used Semmle's LGTM.com and CodeQL to find the others. It is a plain `memcpy` overflow with an attacker-controlled size coming from the network packet without any validation.

The problem exists in the `nfs_readlink_reply` function that parses an nfs reply coming from the network. It parses 4 bytes and, without any further validation, it uses them as length for a `memcpy` in two different locations.

```
static int nfs_readlink_reply(uchar *pkt, unsigned len)
{
        [...]

        /* new path length */
        rlen = ntohl(rpc_pkt.u.reply.data[1 + nfsv3_data_offset]);

        if (*((char *)&(rpc_pkt.u.reply.data[2 + nfsv3_data_offset])) != '/') {
                int pathlen;

                strcat(nfs_path, "/");
                pathlen = strlen(nfs_path);
                memcpy(nfs_path + pathlen,
                       (uchar *)&(rpc_pkt.u.reply.data[2 + nfsv3_data_offset]),
                       rlen);
                nfs_path[pathlen + rlen] = 0;
```

```
        } else {
                memcpy(nfs_path,
                       (uchar *)&(rpc_pkt.u.reply.data[2 + nfsv3_data_offset]),
                       rlen);
                nfs_path[rlen] = 0;
        }
        return 0;
}
```

The destination buffer `nfs_path` is a global one that can hold up to 2048 bytes.

Variant Analysis using CodeQL

We used the following query that gave us a very manageable list of 9 results to follow up manually. The idea behind the query is to perform a data flow analysis from any helper functions such as `ntohl()`/`ntohs()`/…to the size argument of `memcpy`.

```
import cpp

import semmle.code.cpp.dataflow.TaintTracking
import semmle.code.cpp.rangeanalysis.SimpleRangeAnalysis

class NetworkByteOrderTranslation extends Expr {
  NetworkByteOrderTranslation() {
    // On Windows, there are ntoh* functions.
    this.(Call).getTarget().getName().regexpMatch("ntoh(l|ll|s)")
    or
    // On Linux, and in some code bases, these are defined as macros.
    this = any(MacroInvocation mi |
        mi.getOutermostMacroAccess().getMacroName().regexpMatch("(?i)(^|.*_)ntoh(l|ll|s)")
      ).getExpr()
  }
}

class NetworkToMemFuncLength extends TaintTracking::Configuration {
  NetworkToMemFuncLength() { this = "NetworkToMemFuncLength" }

  override predicate isSource(DataFlow::Node source) {
       source.asExpr() instanceof NetworkByteOrderTranslation
  }

  override predicate isSink(DataFlow::Node sink) {
    exists (FunctionCall fc |
        fc.getTarget().getName().regexpMatch("memcpy|memmove") and
        fc.getArgument(2) = sink.asExpr() )
  }

}

from Expr ntoh, Expr sizeArg, NetworkToMemFuncLength config
where config.hasFlow(DataFlow::exprNode(ntoh), DataFlow::exprNode(sizeArg))
select ntoh.getLocation(), sizeArg
```

Did we find any variants?

We went through the results and while some have the size checked in between the data flow from source to sink, some were found to be exploitable. Additionally, we found some other variants through source code review.

Unbound memcpy with a failed length check at `nfs_lookup_reply`

This problem exists in the `nfs_lookup_reply` function that again parses an nfs reply coming from the network. It parses 4 bytes and uses them as length for a `memcpy` in two different locations.

A length check happens to make sure it is not bigger than the allocated buffer. Unfortunately, this check can be bypassed with a negative value that would lead later to a large buffer overflow.

```
filefh3_length = ntohl(rpc_pkt.u.reply.data\[1]);
if (filefh3_length > NFS3_FHSIZE)
    filefh3_length  = NFS3_FHSIZE;

memcpy(filefh, rpc_pkt.u.reply.data + 2, filefh3_length);
```

The destination buffer `filefh` is a global one that can hold up to 64 bytes.

Unbound memcpy with a failed length check at `nfs_read_reply/store_block`

This problem exists in the `nfs_read_reply` function when reading a file and storing it into another medium (flash or physical memory) for later processing. Again, the data and length is fully controlled by the attacker and never validated.

```
static int nfs_read_reply(uchar *pkt, unsigned len)
{        [...]

        if (supported_nfs_versions & NFSV2_FLAG) {
                rlen = ntohl(rpc_pkt.u.reply.data[18]); // <-- rlen is attacker-controlled could be 0xFFFFFFFF
                data_ptr = (uchar *)&(rpc_pkt.u.reply.data[19]);
        } else {  /* NFSV3_FLAG */
                int nfsv3_data_offset =
                        nfs3_get_attributes_offset(rpc_pkt.u.reply.data);

                /* count value */
                rlen = ntohl(rpc_pkt.u.reply.data[1 + nfsv3_data_offset]); // <-- rlen is attacker-controlled
                /* Skip unused values :
                        EOF:            32 bits value,
                        data_size:      32 bits value,
                */
                data_ptr = (uchar *)
                        &(rpc_pkt.u.reply.data[4 + nfsv3_data_offset]);
        }

        if (store_block(data_ptr, nfs_offset, rlen)) // <-- We pass to store_block source and length controlled by the attacker
                        return -9999;

        [...]
}
```

Focusing on physical memory part of the `store_block` function, it attempts to reserve some memory using the arch specific function `map_physmem`, ending up calling `phys_to_virt`. As you can see in the x86 implementation, when reserving physical memory it clearly ignores length and gives you a raw pointer without checking if surrounding areas are reserved (or not) for other purposes.

```
static inline void *phys_to_virt(phys_addr_t paddr)
{
        return (void *)(unsigned long)paddr;
}
```

Later at `store_block` there is a `memcpy` buffer overrun with attacker-controlled source and length.

```
static inline int store_block(uchar *src, unsigned offset, unsigned len)
{

[...]

 void *ptr = map_sysmem(load_addr + offset, len); // <-- essentially this is ptr = load_addr + offset
 memcpy(ptr, src, len); // <-- unrestricted overflow happens here
 unmap_sysmem(ptr);

[...]

}
```

Potentially, similar problems may exist with the `flash_write` code path.

Unbound memcpy when parsing a UDP packet due to integer underflow

The function `net_process_received_packet` is subject to an integer underflow when using `ip->udp_len` without validation. Later this field is used in a `memcpy` at `nc_input_packet` and any udp packet handlers that are set via `net_set_udp_handler` (DNS, dhcp, …).

```
#if defined(CONFIG_NETCONSOLE) && !defined(CONFIG_SPL_BUILD)
                nc_input_packet((uchar *)ip + IP_UDP_HDR_SIZE,
                                src_ip,
                                ntohs(ip->udp_dst),
                                ntohs(ip->udp_src),
                                ntohs(ip->udp_len) - UDP_HDR_SIZE); // <- integer underflow
#endif
                /*
                 * IP header OK.  Pass the packet to the current handler.
                 */
                (*udp_packet_handler)((uchar *)ip + IP_UDP_HDR_SIZE,
                                      ntohs(ip->udp_dst),
                                      src_ip,
                                      ntohs(ip->udp_src),
                                      ntohs(ip->udp_len) - UDP_HDR_SIZE); // <- integer underflow
```

Please note, we did not audit all potential udp handlers that are set for different purposes (DNS, DHCP, …). However, we did fully audit the `nfs_handler`, as discussed below.

Multiple stack-based buffer overflow in `nfs_handler` reply helper functions

This is a code review variant of the above vulnerability. Here, the integer underflows when parsing a udp packet with a large `ip->udp_len` later calling the `nfs_handler`. In this function, again there is no validation of the length and we call helper functions such as `nfs_readlink_reply`. This function blindly uses the length without validation, causing a stack-based buffer overflow.

```
static int nfs_readlink_reply(uchar *pkt, unsigned len)
{
    struct rpc_t rpc_pkt;

    [...]

        memcpy((unsigned char *)&rpc_pkt, pkt, len);
```

We identified 5 different vulnerable functions subject to the same code pattern, which leads to a stack-based buffer overflow. In addition to `nfs_readlink_reply`:

- `rpc_lookup_reply`
- `nfs_mount_reply`
- `nfs_umountall_reply`
- `nfs_lookup_reply`

Read out-of-bound data at `nfs_read_reply`

This is very similar to the previous vulnerabilities.The developers have tried to be careful by performing size checks while copying the data that come from the socket. While they checked to prevent the buffer overflow they did not check there was enough data in the source buffer, leading to a potential read out-of-bounds access violation.

```
static int nfs_read_reply(uchar *pkt, unsigned len)
{
    struct rpc_t rpc_pkt;

    [...]

        memcpy(&rpc_pkt.u.data[0], pkt, sizeof(rpc_pkt.u.reply));
```

An attacker could supply an NFS packet with a read request and with a small packet request sent to the socket.

Any recommendations?

In order to mitigate these vulnerabilities, there are only two options:

- Apply patches as soon as they are released, or
- While vulnerable, do not use mounting filesystems via NFS or any U-Boot networking functionality

Disclosure timeline

This vulnerability report was subject to our disclosure policy available at https://lgtm.com/security/#disclosure_policy

- May 15, 2019 - Fermín Serna initially finds two vulnerabilities and writes a query that uncovers three more problematic call sites.
- May 16, 2019 - Pavel Avgustinov brings some CodeQL magic, generalizes the query, and finds some more parsing ip and udp headers.
- May 23, 2019 - Kevin Backhouse alerts Pavel and Fermín about an oversight regarding a stack-based buffer overflow via nfs_handler.
- May 23, 2019 - Semmle security team concludes the investigation and contacts maintainers via email.
- May 24, 2019 - Tom Rini (U-Boot's master custodian) acknowledges receiving the security report.
- July 19, 2019 - Tom Rini requests to make this report public at their public mailing list u-boot@lists.denx.de.
- July 22, 2019 - To avoid a weekend disclosure, Fermin makes the report public at u-boot@lists.denx.de.

- Docs
- Community Forum
- Professional Services
- Status
- Contact GitHub

Company

- About
- Blog
- Careers
- Press
- Shop