Talos Vulnerability Report

TALOS-2021-1249

Microsoft Azure Sphere Linux namespace ptrace unsigned code execution vulnerability

APRIL 13, 2021

CVE NUMBER

CVE-2021-27074

Summary

An unsigned code execution vulnerability exists in the Linux namespace ptrace functionality of Microsoft Azure Sphere 21.01. Specially crafted shellcode could allow an adversary to execute unsigned code. An attacker can change the namespace and use ptrace to modify the code of a running process to trigger this vulnerability.

Tested Versions

Microsoft Azure Sphere 21.01

Product URLs

https://azure.microsoft.com/en-us/services/azure-sphere/

CVSSv3 Score

6.2 - CVSS:3.0/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:N

CWE

CWE-284 - Improper Access Control

Details

Microsoft's Azure Sphere is a platform for the development of internet-of-things applications. It features a custom SoC that consists of a set of cores that run both high-level and real-time applications, enforces security and manages encryption (among other functions). The high-level applications execute on a custom Linux-based OS, with several modifications to make it smaller and more secure, specifically for IoT applications.

A Linux namespace is an abstraction provided by the kernel to limit the execution context of a given process or thread and potentially also to isolate it.
Currently, there exist 8 kinds of namespaces: Cgroup, IPC, Network, Mount, PID, Time, User, UTS. An unprivileged user can create a new user namespace (using the `CLONE_NEWUSER` flag) and have a full capabilities (root user with all caps) in that namespace. From `man user_namespaces(7)`:

```
User namespaces isolate security-related identifiers and
attributes, in particular, user IDs and group IDs (see
credentials(7)), the root directory, keys (see keyrings(7)), and
capabilities (see capabilities(7)).  A process's user and group
IDs can be different inside and outside a user namespace.  In
particular, a process can have a normal unprivileged user ID
outside a user namespace while at the same time having a user ID
of 0 inside the namespace; in other words, the process has full
privileges for operations inside the user namespace, but is
unprivileged for operations outside the namespace.
```

Once in the new user namespace, the user has full capabilities in the namespace (including `CAP_SYS_PTRACE`) to all other resources within the namespace, which allows for tracing arbitrary processes within the namespace. It is worth emphasizing again that the elevated permissions only apply to resources within the user namespace, if one were to attempt tracing a process outside the new user namespace, the credentials utilized by the tracer would be the same unpriviledged credentials as before.

In the Azure Sphere platfom, a more novel approach at preventing unsigned code execution exists: only the code already present in the device, or signed code that has been deployed to the device via the cloud, and marked as executable can ever be executed. This is enforced by Linux kernel patches around and inside `mprotect` and `mmap` that make sure that memory that has ever been writable cannot ever be executable. Moreover, this is also enforced at the kernel driver level by ensuring all mountpoints are exclusively either writable or executable.

In order to stop attackers from simply using `PTRACE_ATTACH` and `PTRACE_POKETEXT` to write to non-writable memory for subsequent execution, kernel patches have been added to deny the ptracing of any process unless the device is running in development mode. This is implemented via LSM hooks:

```
static struct security_hook_list azure_sphere_hooks[] = {
    ...
    LSM_HOOK_INIT(ptrace_access_check, azure_sphere_ptrace_access_check),
    LSM_HOOK_INIT(ptrace_traceme, azure_sphere_ptrace_traceme),
    ...
};


// LSM entry only called on ptrace_traceme
static int azure_sphere_ptrace_traceme(struct task_struct *parent)
{
    // if pluton says we are in development mode then allow otherwise fail
    if(azure_sphere_in_dev_mode()) {
        return 0;
    }

    return -EPERM;
}

// LSM entry only called on ptrace_attach and a small subset of /proc entries impacting if you can see other PIDs
static int azure_sphere_ptrace_access_check(struct task_struct *child, unsigned int mode)
{
    struct mm_struct *mm;
    const struct cred *cred = current_cred();
    bool ret;
    struct azure_sphere_task_cred *child_tsec;
    struct azure_sphere_task_cred *self_tsec;

    // if CAP_SYS_PTRACE is active then allow        // [1]
    rcu_read_lock();
    ret = az_ptrace_has_cap(cred, __task_cred(child)->user_ns, mode);
    rcu_read_unlock();
    if (ret) {
        // check if the user_ns is accessible for the task memory
        mm = child->mm;
        if (mm && az_ptrace_has_cap(cred, mm->user_ns, mode)) {
                return 0;
        }
    }

    // make sure that the capabilities of the process is a superset of the process being traced to avoid elevating privileges
    child_tsec = get_task_cred(child)->security;
    self_tsec = get_task_cred(current)->security;
    if(!child_tsec || !self_tsec || ((child_tsec->capabilities & self_tsec->capabilities) != child_tsec->capabilities))
        return -EPERM;

    // if pluton says we are in development mode then allow otherwise fail
    if(!azure_sphere_in_dev_mode()) {
        return -EPERM;
    }

    return 0;
}
```

Notice how `azure_sphere_ptrace_traceme` is disallowed altogether in non-dev mode, while `azure_sphere_ptrace_access_check` (hit from `PTRACE_ATTACH`, `PTRACE_SEIZE` and `PTRACE_MODE_READ`) allows ptracing to occur when not in development mode, assuming process owns the `CAP_SYS_PTRACE` [1] capability.
Indeed, the function `az_ptrace_has_cap` determines if a process is allowed to trace, just by checking `CAP_SYS_PTRACE`:

```
static bool az_ptrace_has_cap(const struct cred *cred, struct user_namespace *ns,
              unsigned int mode)
{
   int ret;

   if (mode & PTRACE_MODE_NOAUDIT)
       ret = security_capable(cred, ns, CAP_SYS_PTRACE, CAP_OPT_NOAUDIT);
   else
       ret = security_capable(cred, ns, CAP_SYS_PTRACE, CAP_OPT_NONE);

   return ret == 0;
}
```

Returning back to `azure_sphere_access_check`:

```
...
ret = az_ptrace_has_cap(cred, __task_cred(child)->user_ns, mode); // [2]
rcu_read_unlock();
if (ret) {
    // check if the user_ns is accessible for the task memory
    mm = child->mm;
    if (mm && az_ptrace_has_cap(cred, mm->user_ns, mode)) {       // [3]
            return 0;
    }
}
```

At [2], the security module checks to see that the tracing process has `CAP_SYS_PTRACE` within the tracee process's user namespace. This check can be passed simply by entering a new username space, `fork()`'ing or `clone()`'ing and then having the parent process ptrace the child process. Since the credentials of the parent are inherited by the child (most importantly the user namespace), and also because the parent has `CAP_SYS_PTRACE` within this user namespace, we can pass this check rather easily. The check at [3] however is not automatically passed with `fork()` or `clone()`, while the child's credential structure might live in the same namespace as the parent, the child's memory map (`child->mm->user_ns`) does not necessarily follow the trend, extra steps must be taken by an attacker (one of the simpler ways to pass this check is to just `execl()` after `clone()`or `fork()`).

Regardless, since there are no checks are in place to verify that the parent process' namespace is not the root namespace (`init_user_ns`), it's allowed to create a new namespace (either via `unshare` or `clone`) and use ptrace inside it. Thus, once an attacker has passed both `az_ptrace_has_cap` checks, it follows that a process can spawn a new child, attach to it via `PTRACE_ATTACH`, modify its executable memory via `PTRACE_POKETEXT`, and let it resume, effectively running unsigned code on the device (similarly to how it was demonstrated in TALOS-2020-1090).

Timeline

2021-02-12 - Vendor Disclosure
2021-04-13 - Public Release

CREDIT

Discovered by Claudio Bozzato and Lilith >_> of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT TALOS-2021-1250

NEXT REPORT TALOS-2021-1262