Code injection on macOS

knight.sc/malware/2019/03/15/code-injection-on-macos.html

March 15, 2019

I was recently reviewing the MITRE ATT&CK™ knowledge base and came across the page on process injection techniques for privilege escalation. For those that are not aware of what the MITRE ATT&CK™ knowledge base is, it's a group of documents and definitions that cover common adversary tactics and techniques. The macOS and Linux sections for process injection were lumped together and not very detailed. In some cases it seemed like the information wasn't even accurate for macOS. This article covers common process injection techniques that apply to macOS.

DYLD_INSERT_LIBRARIES

This is one of the most well known and common techniques for code injection on macOS. By setting the `DYLD_INSERT_LIBRARIES` environment variable to a dylib of their choice and then starting an application an attacker can get the dylib code running inside of the started process. In older versions of macOS this could be used to inject a dylib into an Apple platform application with higher privileges. This would allow the injected dylib to also gain those additional privileges. Since the addition of SIP in macOS 10.12 this technique can no longer be used on Apple platform binaries. As of macOS 10.14 third party developers can also opt in to a hardened runtime for their application. This can also prevent the injection of dylibs using this technique.

Below are a few examples of how DYLD_INSERT_LIBRARIES works on macOS:

http://thomasfinch.me/blog/2015/07/24/Hooking-C-Functions-At-Runtime.html
https://blog.timac.org/2012/1218-simple-code-injection-using-dyld_insert_libraries/

Thread Injection

If you look up code injection techniques on Windows, thread injection is one of the most common. With APIs like `CreateRemoteThread` the entire process is fairly straight forward and doesn't take much code. If you try searching for the same thing on macOS you'll find a lot less resources. Luckily, Jonathan Levin, author of the great MacOS and iOS Internals collection of books has a great example on his website.

http://newosxbook.com/src.jl?tree=listings&file=inject.c

This example makes use of the Mach `thread_create_running` API. Since macOS has a dual personality, with low level Mach APIs as well as BSD APIs, there exists two sets of APIs for working with threads. One is the Mach APIs and the other is the `pthread` APIs. Unfortunately some internal parts of macOS expect every thread to have been properly

created from the BSD APIs and to have all Mach thread structures as well as `pthread` structures set up properly. In order to handle this, the inject.c example above, attempts to first call `_pthread_set_self` in the injected code in order to get the thread to a working state.

This approach works well up to macOS 10.14 where some of the `pthread` internal code changed. I wanted to get a working version of this example on 10.14 and up so I decided to look into some of the `pthread` code. Prior to macOS 10.14, the `_pthread_set_self` code did the following:

libpthread-301.50.1/src/pthread.c

```
PTHREAD_NOINLINE
void
_pthread_set_self(pthread_t p)
{
        return _pthread_set_self_internal(p, true);
}

PTHREAD_ALWAYS_INLINE
static inline void
_pthread_set_self_internal(pthread_t p, bool needs_tsd_base_set)
{
        if (p == NULL) {
                p = &_thread;
        }

        uint64_t tid = __thread_selfid();
        if (tid == -1ull) {
                PTHREAD_ABORT("failed to set thread_id");
        }

        p->tsd[_PTHREAD_TSD_SLOT_PTHREAD_SELF] = p;
        p->tsd[_PTHREAD_TSD_SLOT_ERRNO] = &p->err_no;
        p->thread_id = tid;

        if (needs_tsd_base_set) {
                _thread_set_tsd_base(&p->tsd[0]);
        }
}
```

This code allows us to pass `NULL` into the `_pthread_set_self` call and in turn it will set up some of the internal `pthread` structures based on the main thread of the application. This is ideal in the injection case because we're starting from a bare Mach thread with no `pthread` structures set up and no reference to any other thread. On macOS 10.14 and higher this code has changed and you can no longer pass `NULL` into `_pthread_set_self`

libpthread-330.201.1/src/pthread.c

```
PTHREAD_NOINLINE
void
_pthread_set_self(pthread_t p)
{
#if VARIANT_DYLD
        if (os_likely(!p)) {
                return _pthread_set_self_dyld();
        }
#endif // VARIANT_DYLD
        _pthread_set_self_internal(p, true);
}

#if VARIANT_DYLD
// _pthread_set_self_dyld is noinline+noexport to allow the option for
// static libsyscall to adopt this as the entry point from mach_init if
// desired
PTHREAD_NOINLINE PTHREAD_NOEXPORT
void
_pthread_set_self_dyld(void)
{
        pthread_t p = main_thread();
        p->thread_id = __thread_selfid();

        if (os_unlikely(p->thread_id == -1ull)) {
                PTHREAD_INTERNAL_CRASH(0, "failed to set thread_id");
        }

        // <rdar://problem/40930651> pthread self and the errno address are the
        // bare minimium TSD setup that dyld needs to actually function.  Without
        // this, TSD access will fail and crash if it uses bits of Libc prior to
        // library initialization. __pthread_init will finish the initialization
        // during library init.
        p->tsd[_PTHREAD_TSD_SLOT_PTHREAD_SELF] = p;
        p->tsd[_PTHREAD_TSD_SLOT_ERRNO] = &p->err_no;
        _thread_set_tsd_base(&p->tsd[0]);
}
#endif // VARIANT_DYLD

PTHREAD_ALWAYS_INLINE
static inline void
_pthread_set_self_internal(pthread_t p, bool needs_tsd_base_set)
{
        p->thread_id = __thread_selfid();

        if (os_unlikely(p->thread_id == -1ull)) {
                PTHREAD_INTERNAL_CRASH(0, "failed to set thread_id");
        }

        if (needs_tsd_base_set) {
                _thread_set_tsd_base(&p->tsd[0]);
        }
}
```

The internal implementation was split into a dyld specific one not accessible in the user space `libpthread` library and the other internal one which expects a valid thread to be passed in. In fact `_pthread_set_self_internal` will crash if null is passed in because it expects the argument to be there.

I decided to continue reviewing the `pthread` source code to look for another function that could help bootstrap a bare Mach thread into a properly set up `pthread`. I ended up coming across the `pthread_create_from_mach_thread` function. This function has existed since macOS 10.12 so it should work on 10.12 and up. It calls into the internal `_pthread_create` implementation passing in `true` to the `from_mach_thread` argument. I could only find one binary on my system that actually used this API: `RemoteInjectionAgent` within the Xcode `DVTInstrumentsFoundation.framework`.

The idea is to inject a bare Mach thread as a bootstrap thread and then use the `pthread_create_from_mach_thread` to create a second fully configured, legitimate `pthread`. Here's the modified `injectedCode` from Jonathan Levin's example.

```
                    _injectedCode:
00000001000020d0        push    rbp                                     ;
DATA XREF=_inject+576, _inject+1014
00000001000020d1        mov     rbp, rsp
00000001000020d4        sub     rsp, 0x10
00000001000020d8        lea     rdi, qword [rbp-8]
00000001000020dc        xor     eax, eax
00000001000020de        mov     ecx, eax
00000001000020e0        lea     rdx, qword [_injectedCode+56]           ;
0x100002108
00000001000020e7        mov     rsi, rcx
00000001000020ea        movabs  rax, 0x5452434452485450                 ;
PTHRDCRT
00000001000020f4        call    rax
00000001000020f6        mov     dword [rbp-0xc], eax
00000001000020f9        add     rsp, 0x10
00000001000020fd        pop     rbp
00000001000020fe        mov     rax, 0xd13
0000000100002105        jmp     _injectedCode+53                        ;
CODE XREF=_injectedCode+53
0000000100002107        ret

0000000100002108        push    rbp                                     ;
DATA XREF=_injectedCode+16
0000000100002109        mov     rbp, rsp
000000010000210c        sub     rsp, 0x10
0000000100002110        mov     esi, 0x1
0000000100002115        mov     qword [rbp-8], rdi
0000000100002119        lea     rdi, qword [aLiblibliblib]              ;
"LIBLIBLIBLIB"
0000000100002120        movabs  rax, 0x5f5f4e45504f4c44                 ;
DLOPEN__
000000010000212a        call    rax
000000010000212c        xor     esi, esi
000000010000212e        mov     edi, esi
0000000100002130        mov     qword [rbp-0x10], rax
0000000100002134        mov     rax, rdi
0000000100002137        add     rsp, 0x10
000000010000213b        pop     rbp
000000010000213c        ret
                    aLiblibliblib:
000000010000213d        db      "LIBLIBLIBLIB", 0                       ;
DATA XREF=_injectedCode+73
```

You can download a full updated working example of this code from the link below:

https://gist.github.com/knightsc/45edfc4903a9d2fa9f5905f60b02ce5a

There's a couple notes on this technique. First it depends on being able to call `task_for_pid` to get the Mach task port of the victim process. You can only do this as root and just like dylib injection you can not use `task_for_pid` on Apple platform binaries due

to SIP on macOS 10.12 and higher. So while it's still an interesting technique it's not as useful for privilege escalation. This technique has been used in the past in iOS exploits in cases where another exploit has allowed a task port to be leaked over to an attacker process.

Thread Hijacking

Another possible techinque on macOS is thread hijacking. Instead of creating a thread in a remote process we instead retrieve an existing thread and coerce it into running what we want. Apple has continued to lock down `task_for_pid` as well as any Mach API that takes a task port in order to try to prevent the abuse of leaked task ports. Due to this, thread hijacking has becomes a more interesting technique. Brandon Azad has an amazing write up around this technique and I'm not going to attempt to cover it in great detail here. I highly recommend you go and read the following:

https://bazad.github.io/2018/10/bypassing-platform-binary-task-threads/

I looked into this technique briefly and attempted to hijack a thread, run code and then put the thread back to its original state. It appears that what we can save with `thread_get_state` doesn't really save all of the state and the thread often crashes. It's good enough for other uses though if you're just trying to execute code in the context of a privileged app but not good enough if you're trying to take control of another process without notice. You can see my code example here:

https://gist.github.com/knightsc/bd6dfeccb02b77eb6409db5601dcef36

If you're interested in this technique I highly recommend reading over the code to Brandon Azad's threadexec library. It goes into great detail around this technique and goes along with his article above. Unfortunately it seems like he came to a similar conclusion as me in that trying to save and restore the thread state does not work that reliably.

ptrace?

If you read the ATT&CK page you might have been led to believe that on Linux and macOS the `ptrace` APIs could be used for code injection. That's not actually the case on macOS. While the `ptrace` syscall does exist on macOS it is not fully implemented. For instance none of the `PTRACE_PEEKTEXT`, `PTRACE_POKETEXT`, `PTRACE_GETREGS`, `PTRACE_SETREGS` calls exist.

Other techniques?

I think there could also exist other techniques that haven't been explored yet. With `libdispatch` being one of the core libraries enabling applications to do work in parallel it seems like that might be an area that hasn't fully been explored yet. My thought is that it might be possible to inject code into a remote process that is in the format of a valid dispatch

block and then get that block submitted to a work queue. Alternatively it might be possible to locate a block queued up but not currently running and hijack the code that the block points too. I haven't yet had time to dig into this more but I think it's definitely an interesting area of research.

Conclusion

Hopefully you can see that there are a wide variety of different techniques for code injection on macOS. My hope is that with more articles like this the knowledge will continue to spread. In my next article I plan on covering different ways that some of these techniques can be detected.