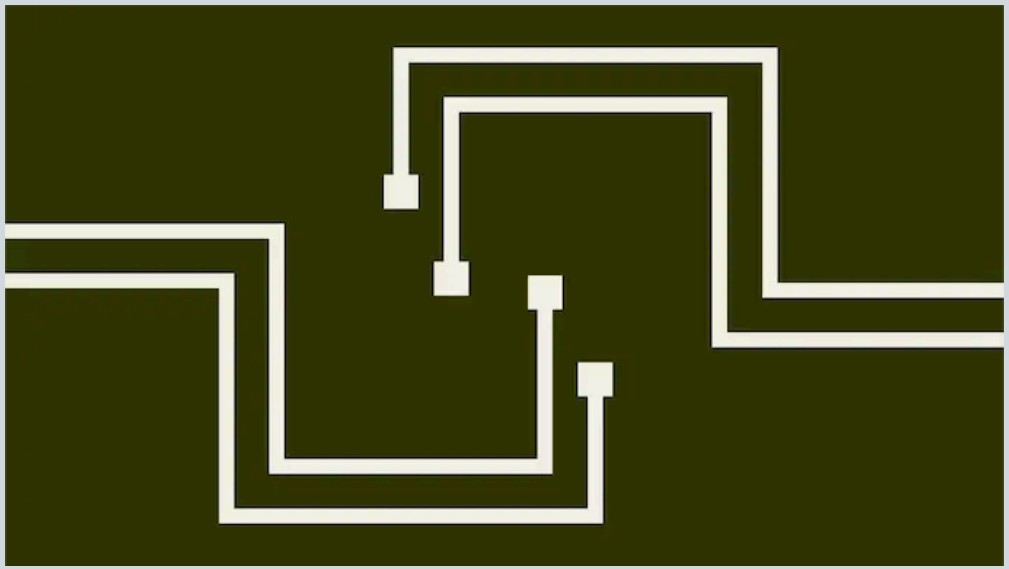


Adversary Simulation | August 23, 2017

sRDI – Shellcode Reflective DLL Injection

Nick Landers

During our first offering of “Dark Side Ops II – Adversary Simulation” at Black Hat USA 2017, we quietly dropped a piece of our internal toolkit called sRDI. Shortly after, the full project was put on GitHub (**https://github.com/monoxgas/sRDI**) without much explanation. I wanted to write a quick post discussing the details and use-cases behind this new functionality.

A Short History

Back in ye olde times, if you were exploiting existing code, or staging malicious code into memory, you used shellcode. For those rare few who still have the skill to write programs in assembly, we commend you. As the Windows API grew up and gained popularity, people found sanctuary in DLLs. C code and cross compatibility were very appealing, but what if you wanted your DLL to execute in another process? Well, you could try writing the file to memory and dropping a thread at the top, but that doesn't work very well on **packed PE files**. The Windows OS already knows how to load PE files, so people asked nicely and **DLL Injection** was born. This involves starting a thread in a remote process to call "LoadLibrary()" from the WinAPI. This will read a (malicious) DLL from disk and load it into the target process. So you write some cool malware, save it as a DLL, drop it to disk, and respawn into other processes. Awesome!.well, not really. Anti-virus vendors caught on quick, started flagging more and more file types, and performing heuristic analysis. The disk wasn't a safe place anymore!

Finally in 2009, our malware messiah Stephen Fewer (**@stephenfewer**) releases **Reflective DLL Injection**. As demonstrated, LoadLibrary is limited in loading only DLLs from disk. So Mr. Fewer said "Hold my beer, I'll do it myself". With a rough copy of LoadLibrary implemented in C, this code could now be included into any DLL project. The process would export a new function called "ReflectiveLoader" from the (malicious) DLL. When injected, the reflective DLL would locate the offset of this function, and drop a thread on it. ReflectiveLoader walks back through memory to locate the beginning of the DLL, then unpacks and remaps everything automatically. When complete, "DLLMain" is called and you have your malware running in memory.

Years went by and very little was done to update these techniques. Memory injection was well ahead of it's time and allowed all the APTs and such to breeze past AV. In 2015, Dan Staples (**@_dismantl**) released an important update to RDI, called "**Improved Reflective DLL Injection**". This aimed to allow an additional function to be called after "DLLMain" and support the passing of user arguments into said additional function. Some shellcode trickery and a bootstrap placed before the call to

ReflectiveLoader accomplished just that. RDI is now functioning more and more like the legitimate LoadLibrary. We can now load a DLL, call it's entry point, and then pass user data to **another** exported function. By the way, if you aren't familiar with DLLs or exported functions, I recommend you read **Microsoft's overview**.

Making shellcode great again

Reflective DLL injection is being used heavily by private and public toolsets to maintain that "in-memory" street cred. Why change things? Well.

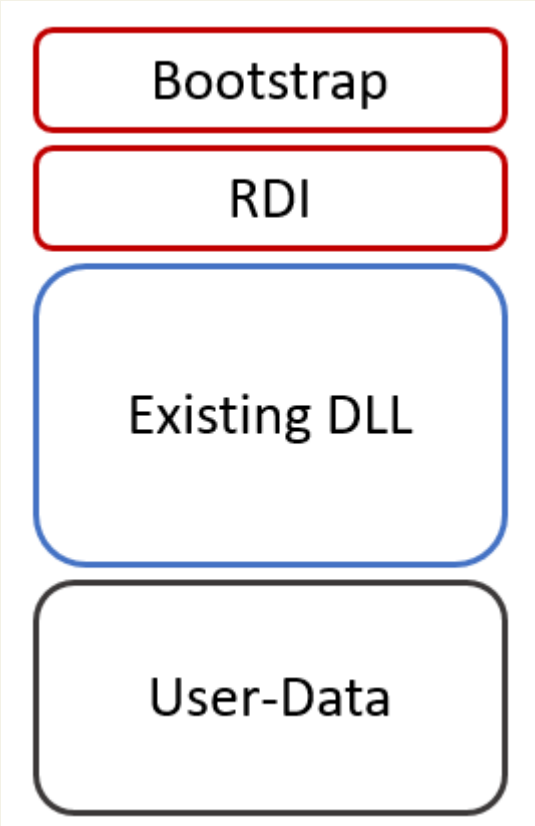
- RDI requires that your target DLL and staging code **understand** RDI. So you need access to the source code on both ends (the injector and injectee), or use tools that already support RDI.
- RDI requires a lot of code for loading in comparison to shellcode injection. This compromises stealth and makes stagers easier to signature/monitor.
- RDI is confusing for people who don't write native code often.
- Modern APT groups have already implemented more mature **memory injection techniques**, and our goal is better emulate real-world adversaries.

The list isn't as long as some reasons to change things, but we wanted to write a new version of RDI for simplicity and flexibility. So what did we do?

1. To start, we read through some great research by Matt Graeber (**@mattifestation**) to **convert primitive C code into shellcode.** We rewrote the ReflectiveLoader function and converted the entire thing into a big shellcode blob. We now have a basic PE loader as shellcode.

2. We wanted to maintain the advantages of Dan Staples technique, so we modified the bootstrap to hook into our new shellcode ReflectiveLoader. We also added some other tricks like a pop/call to allow the shellcode to get it's current location in memory and maintain position independence.
3. Once our bootstrap primitives were built, we implemented a conversion process into different languages (C, PowerShell, C#, and Python). This allows us to hook our new shellcode and a DLL together with the bootstrap code in any other tool we needed.

Once complete, the blob looks something like this:



When execution starts at the top of the bootstrap, the general flow looks like this:

1. Get current location in memory (Bootstrap)
2. Calculate and setup registers (Bootstrap)
3. Pass execution to RDI with the function hash, user data, and location of the target DLL (Bootstrap)
4. Un-pack DLL and remap sections (RDI)
5. Call DLLMain (RDI)
6. Call exported function by hashed name (RDI) – Optional
7. Pass user-data to exported function (RDI) – Optional

With that all done, we now have conversion functions that take in arbitrary DLLs, and spit out position independent shellcode. Optionally, you can specify arbitrary data to get passed to an exported function once the DLL is loaded (as Mr. Staples intended). On top of that, if you are performing local injection, the shellcode will return a memory pointer that you can use with GetProcAddressR() to locate additional exported functions and call them. Even with the explanation, the process can seem confusing to most who don't have experience with the original RDI project, shellcode, or PE files, so I recommend you read existing research and head over to the GitHub repository and dig into the code: **https://github.com/monoxgas/sRDI**

Okay, so what?

"You can now convert any DLL to position independent shellcode at any time, on the fly."

This tool is mainly relevant to people who write/customize malware. If you don't know how to write a DLL, I doubt most of this applies to you. With that said, if you are interested in writing something more than a PowerShell script or Py2Exe executable to

perform red-teaming, this is a great place to start.

Use case #1 – Stealthy persistence

- Use server-side Python code (sRDI) to convert a RAT to shellcode
- Write the shellcode to the registry
- Setup a scheduled task to execute a basic loader DLL
- Loader reads shellcode and injects (<20 lines of C code)

Pros: Neither your RAT or loader need to understand RDI or be compiled with RDI. The loader can stay small and simple to avoid AV.

Use case #2 – Side loading

- Get your sweet RAT running in memory
- Write DLL to perform extra functionality
- Convert the DLL to shellcode (using sRDI) and inject locally
- Use GetProcAddressR to lookup exported functions
- Execute additional functionality X-times without reloading DLL

Pros: Keep your initial tool more lightweight and add functionality as needed. Load a DLL once and use it just like any other.

Use case #3 – Dependencies

- Read existing legitimate API DLL from disk
- Convert the DLL to shellcode (using sRDI) and load it into memory
- Use GetProcAddress to lookup needed functions

Pros: Avoid monitoring tools that detect LoadLibrary calls. Access API functions without leaking information. (WinInet, PSApi, TlHelp32, GdiPlus)

Conclusion

We hope people get good use out of this tool. sRDI been a member of the SBS family for almost 2 years now and we have it integrated into many of our tools. Please make modifications and create pull-requests if you find improvements.

We'd love to see people start pushing memory injection to higher levels. With recent AV vendors promising more analytics and protections against techniques like this, we're confident threat actors have already implemented improvements and alternatives that don't involve high level languages like PowerShell or JScript.

@monoxgas