sRDI — Shellcode反射式DLL注入技术

Max_hhg 看雪学苑 2023年09月29日 11:59 上海

在我们在2017年的“Black Hat USA”上首次推出“暗面行动II - 对抗模拟”期间，我们悄悄地发布了一个名为sRDI的内部工具。不久之后，整个项目被放在GitHub上（https://github.com/monoxgas/sRDI），但没有给出太多解释。我想写一篇简短的文章，讨论这个新功能的细节和用途。

sRDI是一个Shellcode Reflective DLL Injection的工具，它可以将DLL文件转换为位置无关的shellcode。它具有完整的PE加载器功能，支持正确的节权限、TLS回调和健全性检查。可以将其视为一个绑定了打包的DLL的Shellcode PE加载器。

简史

在过去的年代中，如果你要利用现有的代码或将恶意代码加载到内存中，你会使用shellcode。对于那些仍然具备使用汇编语言编写程序技能的少数人，我们对你们表示赞赏。随着Windows API的发展和普及，人们开始将目光转向了DLL。C语言和跨平台兼容性非常吸引人，但如果你想让你的DLL在另一个进程中执行怎么办呢？嗯，你可以尝试将文件写入内存并在顶部创建一个线程，但这在经过打包的PE文件上效果并不好。Windows操作系统已经知道如何加载PE文件，所以人们礼貌地提出请求，并诞生了DLL注入技术。这涉及在远程进程中启动一个线程，调用WinAPI中的"LoadLibrary()"函数。这将从磁盘中读取一个（恶意的）DLL文件，并将其加载到目标进程中。因此，你可以编写一些很酷的恶意软件，将其保存为DLL文件，将其放到磁盘上，并在其他进程中重新生成。太棒了！...嗯，并不完全是这样。杀毒软件厂商很快就意识到了这一点，开始标记越来越多的文件类型，并进行启发式分析。磁盘不再是一个安全的地方了！
终于在2009年，我们的恶意软件救世主Stephen Fewer（@stephenfewer）发布了反射式DLL注入（Reflective DLL Injection）。正如演示所示，LoadLibrary仅限于从磁盘加载DLL。因此，Fewer先生说：“拿着我的啤酒，我来自己做。”通过在C中实现了一个粗糙的LoadLibrary副本，现在可以将这段代码包含到任何DLL项目中。该进程将从（恶意的）DLL中导出一个名为"ReflectiveLoader"的新函数。在注入时，反射式DLL将定位该函数的偏移量，并在其上创建一个线程。ReflectiveLoader会回溯内存以找到DLL的起始位置，然后自动

解包和重新映射所有内容。完成后，将调用"DLLMain"，于是你的恶意软件就在内存中运行了。

多年过去了，很少有更新这些技术的工作。内存注入技术已经领先于时代，使得所有高级持续性威胁等都能轻松绕过杀毒软件。2015年，Dan Staples（@_dismantl）发布了对RDI的重要更新，称为“改进的反射式DLL注入”。这旨在允许在"DLLMain"之后调用其他函数，并支持将用户参数传递给该额外函数。通过一些shellcode的技巧和在调用ReflectiveLoader之前放置引导代码，就实现了这一点。现在，RDI的功能越来越像合法的LoadLibrary。我们现在可以加载一个DLL，调用其入口点，然后将用户数据传递给另一个导出函数。顺便说一句，如果你对DLL或导出函数不熟悉，我建议你阅读一下微软的概述文档。

让shellcode再次伟大

反射式DLL注入被私人和公共工具集广泛使用，以保持其在内存中的可信度。为什么要改变呢？原因如下：

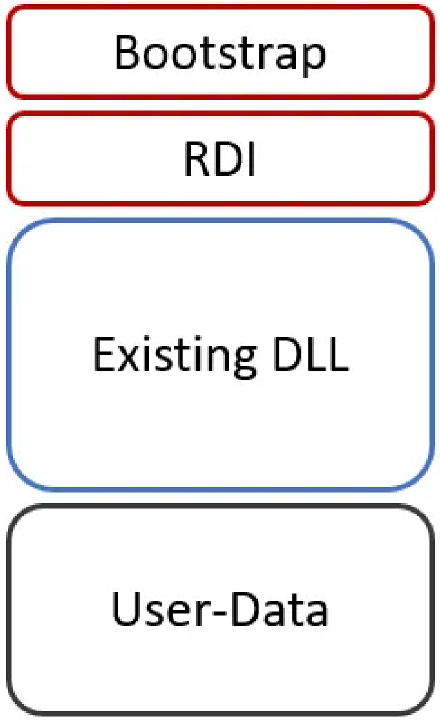
◆RDI要求目标DLL和分段代码都**理解**RDI。因此，你需要在两端（注入器和被注入者）都能够访问源代码，或者使用已经支持RDI的工具。
◆与shellcode注入相比，RDI需要大量的加载代码。这会损害隐蔽性，使分段加载器更容易被签名/监控。
◆对于不经常编写本地代码的人来说，RDI很令人困惑。
◆现代高级持续性威胁组织已经实施了更成熟的内存注入技术，我们的目标是更好地模拟真实的对手。

为了简化和提高灵活性，我们决定编写一种新版本的反射式DLL注入（RDI）。那么我们做了什么呢？

1.首先，我们阅读了Matt Graeber（@mattifestation）的一些优秀研究，将原始的C代码转换为shellcode。我们重写了ReflectiveLoader函数，并将整个函数转换为一个大的shellcode块。现在我们拥有了一个基本的PE加载器作为shellcode。
2.我们希望保留Dan Staples技术的优势，因此我们修改了引导代码，使其与我们的新shellcode ReflectiveLoader连接起来。我们还添加了一些其他技巧，例如使用pop/call来获取shellcode在内存中的当前位置，并保持位置独立性。

3.一旦我们构建了引导代码的基本原语，我们就实现了将其转换为不同语言（C、PowerShell、C#和Python）的过程。这使我们能够将我们的新shellcode和DLL与引导代码一起钩入到任何其他所需的工具中。

完成后，shellcode的代码块可能如下所示：



当执行从引导代码的顶部开始时，大致的流程如下：

- 获取当前内存位置（引导代码）
- 计算和设置寄存器（引导代码）
- 将执行权交给RDI，传递函数哈希、用户数据和目标DLL的位置（引导代码）
- 解包DLL并重新映射各个节（RDI）
- 调用DLLMain（RDI）
- 通过哈希名称调用导出函数（RDI）- 可选
- 将用户数据传递给导出函数（RDI）- 可选

完成了上述工作后，现在我们有了可以接收任意DLL并生成位置无关的shellcode的转换函数。如果需要，您还可以指定要传递给加载完成后导出函数的任意数据（正如Staples先生所期望的）。此外，如果执行本地注入，shellcode将返回一个内存指针，您可以使用GetProcAddressR()函数定位其他导出函数并调用它们。即使在解释说明的情况下，对于没有经验的人来说，这个过程可能仍然很困惑，特别是对于原始RDI项目、shellcode或PE文件没有经验的人。因此，我建议您阅读现有的研究，并访问GitHub存储库并深入了解代码：https://github.com/monoxgas/sRDI

好的，那又怎样呢？

“现在，您可以随时将任何DLL转换为位置无关的shellcode。”
这个工具主要适用于编写/定制恶意软件的人。如果您不知道如何编写DLL，我怀疑大部分内容对您来说并不适用。然而，如果您有兴趣编写比PowerShell脚本或Py2Exe可执行文件更高级的用于红队行动的工具，那么这是一个很好的起点。

使用案例＃1 - 隐蔽的持久性

◆使用服务器端的Python代码（sRDI）将RAT转换为shellcode
◆然后将shellcode写入注册表
◆设置定时任务来执行一个基本的加载器DLL
◆该加载器DLL将读取shellcode并进行注入（不超过20行的C代码）。

优点：您的RAT或加载器都不需要理解RDI或使用RDI进行编译。加载器可以保持小巧简单，以避免被杀毒软件检测。

使用案例＃2 - 侧加载

◆让您的RAT在内存中运行
◆编写DLL以执行额外的功能
◆将DLL转换为shellcode（使用sRDI）并进行本地注入
◆使用GetProcAddressR来查找导出函数
◆无需重新加载DLL，执行额外功能X次

优点：使您的初始工具更轻巧，并根据需要添加功能。只需加载一次DLL，就可以像使用其他任何DLL一样使用它。

使用案例 # 3 - 依赖项

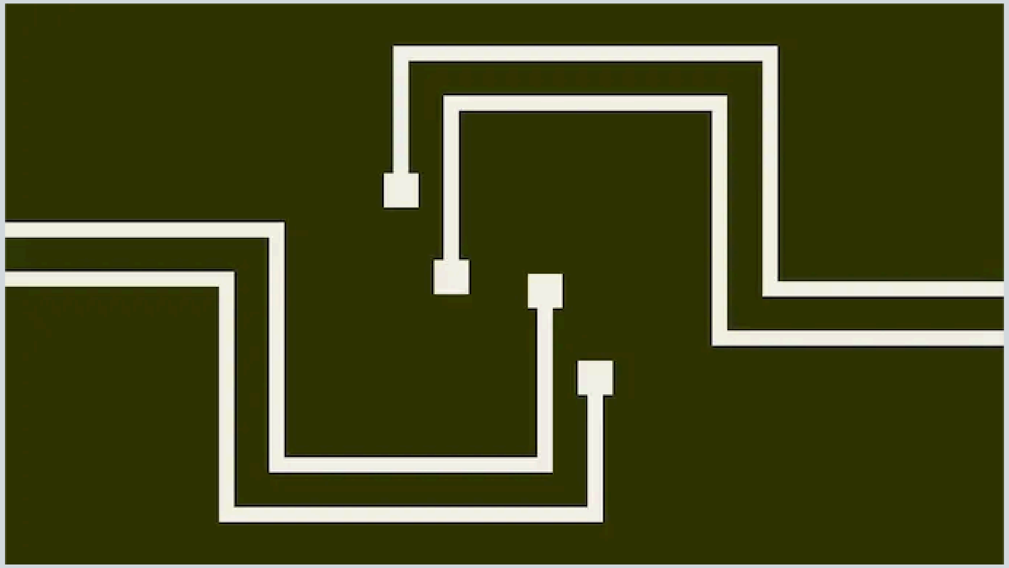
◆从磁盘读取现有的合法API DLL
◆将DLL转换为shellcode（使用sRDI）并加载到内存中
◆使用GetProcAddress来查找所需的函数

优点：避免监控工具检测LoadLibrary调用。在不泄漏信息的情况下访问API函数。（例如WinInet、PSApi、TlHelp32、GdiPlus）

总结

我们希望人们能充分利用这个工具。sRDI已经成为SBS家族的一员将近2年了，我们已将其集成到我们的许多工具中。如果您发现了改进之处，请进行修改并创建拉取请求。

我们希望看到人们开始将内存注入推向更高的水平。随着最近的杀毒软件供应商承诺提供更多分析和对抗此类技术的保护措施，我们相信威胁行为者已经实施了改进和替代方案，而这些方案不涉及像PowerShell或JScript这样的高级语言。



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.

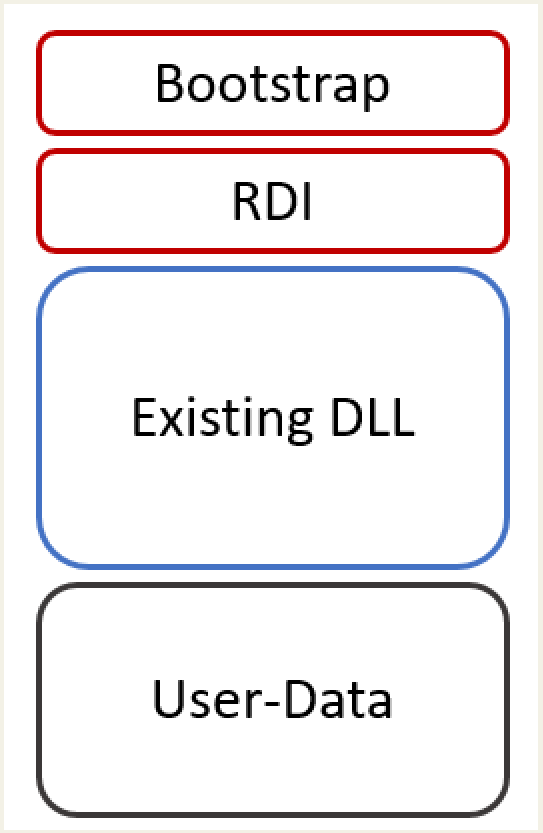
- RDI requires that your target DLL and staging code **understand** RDI. So you need access to the source code on both ends (the injector and injectee), or use tools that already support RDI.
- RDI requires a lot of code for loading in comparison to shellcode injection. This compromises stealth and makes 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