# randomize\_sw2\_seed.py

## get\_old\_seed():

它读取名为'include/syscalls.h'的文件内容，查找包含名为SW2\_SEED的常量的16进制值的特定行，并将该值作为字符串返回。

该函数使用**open()**函数打开文件'include/syscalls.h'，并将其内容读取到变量**code**中。然后，它使用**re.search()**函数在文件的内容中搜索特定的模式。 它搜索的模式是以“＃define SW2\_SEED”开头，并跟随一个8个字符的十六进制字符串（0x [a-fA-F0-9] {8}）的字符串。

如果找到匹配项，则函数使用**match**对象的**group()**方法将匹配的十六进制字符串作为字符串返回。如果没有找到匹配项，则函数引发一个断言错误，并显示消息“SW2\_SEED not found!”。

## replace\_seed

它将名为“include/syscalls.h”的文件中的一行中的旧十六进制种子值替换为新十六进制种子值。

函数使用**open()**函数打开文件“include/syscalls.h”，并将其内容读取到变量**code**中。然后，使用字符串的**replace()**方法将旧十六进制种子值替换为新十六进制种子值。该函数使用字符串格式化和占位符将新种子值转换为16进制字符串，并将其插入到替换字符串中。

函数的第三个参数“1”告诉它只替换一次。这是为了确保不会意外替换其他包含旧种子值的行。

最后，该函数使用**open()**函数再次打开文件“include/syscalls.h”以写模式，并将新的文件内容写回文件中。

## get\_function\_hash

它用于生成给定种子值、函数名称和系统调用类型的函数哈希值。

该函数使用给定的种子值作为初始哈希值，并对给定的函数名称进行一系列哈希操作以生成最终哈希值。函数名称首先被清理，即将所有下划线字符删除。如果is\_syscall参数为True，并且函数名称以“Nt”开头，则函数名称的前缀被替换为“Zw”。

接下来，函数名称被编码为一个字节数组，并在每个2字节的片段中进行哈希处理。这些片段使用小端字节序转换为16位整数，并将它们添加到函数哈希值中。还使用了一个名为“ror8”的lambda函数，该函数用于按位旋转32位哈希值的字节。最终哈希值在每次处理片段时都会被更新。

最后，该函数返回最终的函数哈希值作为一个无符号32位整数。

## replace\_syscall\_hashes

用于将系统调用哈希值替换为给定种子值的哈希值。

该函数首先读取源代码文件“syscalls.c”和“syscalls-asm.asm”的内容。然后，它使用正则表达式搜索“syscalls.c”文件中所有系统调用的名称，并将这些名称存储在一个集合中。

接下来，该函数在“syscalls.c”文件的第五个条件语句块中搜索所有系统调用的定义。它使用正则表达式从每个定义中提取哈希值，并使用get\_function\_hash函数将给定的种子值替换为新的哈希值。然后，它将旧哈希值替换为新哈希值，并将新代码写回到“syscalls.c”文件中。

接下来，该函数读取“syscalls-asm.asm”文件中的内容，并使用正则表达式搜索每个系统调用的汇编代码。它使用类似的方法，使用正则表达式从每个汇编代码中提取哈希值，并将给定的种子值替换为新的哈希值。然后，它将旧哈希值替换为新哈希值，并将新代码写回到“syscalls-asm.asm”文件中。

最后，该函数会打印每个系统调用的旧哈希值和新哈希值，以供参考。如果在搜索过程中找不到哈希值，则会引发断言错误。

## replace\_dinvoke\_hashes

该函数的作用是替换一个系统调用和一组调用的哈希值，从而将其与提供的种子值相匹配。替换哈希值是因为系统调用和调用是根据其哈希值标识的，如果种子值发生更改，则哈希值也必须更改，以便它们仍然正确地识别。为此，函数使用提供的种子值计算新的哈希值，并将其替换为代码中现有的哈希值。对于系统调用，函数使用提供的种子值计算一个新的哈希值，并在包含系统调用的代码中替换现有的哈希值。对于一组调用，函数查找包含哈希值定义的头文件，并将现有哈希值替换为新的哈希值。

# bin2c

用于将二进制文件转换成 C 语言数组。

该程序读取两个命令行参数：输入文件名和数组名。它打开输入文件，并将每个字节作为十六进制数写入 stdout，每个数字之间用逗号分隔。在最后一个字节后，程序会输出一个分号，并输出数组长度的变量声明。

输出的内容类似于以下内容：

arduinoCopy code

#pragma once unsigned char my\_array[] = {0x12,0x34,0x56,0x78,0x9A,0xBC,0xDE,0xF0}; unsigned int my\_array\_len = 8;

你可以在你的程序中包含输出的头文件并使用定义的数组变量来使用二进制数据。

# delete\_file

作用是删除指定路径下的文件。程序包括一个名为 do\_delete 的函数，该函数接受一个文件路径参数，并尝试删除该路径下的文件。如果该文件不存在，则会打印错误信息并返回；如果删除文件失败，则也会打印错误信息并返回；如果成功删除文件，则会打印成功信息并返回。程序使用了 delete\_file.h 中定义的 delete\_file 函数以实现文件删除功能。如果定义了 BOF 宏，则还会包含 utils.c 和 syscalls.c 两个文件，其中 syscalls.c 包含了 Windows 系统调用的实现。如果未定义 BOF 宏，则会在 main 函数中读取命令行参数，并调用 do\_delete 函数来执行文件删除。

# Dinvoke

## is\_dll

用于判断给定的 **HMODULE** 是否表示一个 DLL 模块。这里的 **HMODULE** 实际上是一个指向 DLL 模块的基地址的句柄。

函数首先会检查传入的 **hLibrary** 是否为空，如果为空则直接返回 FALSE，表示不是 DLL 模块。

然后它会将 **hLibrary** 转换成指向 DOS 头的指针，检查 DOS 头的魔数是否为 MZ，如果不是则返回 FALSE，表示不是一个有效的 DOS 头。

接着，它会根据 DOS 头中指定的偏移量获取 NT 头的指针，并检查 NT 头的签名是否为 IMAGE\_NT\_SIGNATURE，如果不是则返回 FALSE，表示不是一个有效的 NT 头。

最后，它会检查 NT 头中的 Characteristics 字段，如果其中包含 IMAGE\_FILE\_DLL 标志，则表示该模块是一个 DLL 模块，返回 TRUE；否则表示它是一个 PE 模块，返回 FALSE。

## find\_legacy\_export

用于在当前进程中查找包含特定导出函数的 DLL。它接受两个参数：hOriginalLibrary 是在该 DLL 中调用此函数的调用方的句柄，fhash 是要查找的导出函数的哈希值。函数首先获取当前进程的 PEB (进程环境块) 并遍历其中的模块列表以查找 DLL。对于每个模块，函数使用 get\_function\_address() 函数查找导出函数地址，如果找到，则返回该地址。如果找不到，则返回 NULL。在查找过程中，如果发现当前模块与调用方 DLL 相同，则跳过该模块。

## resolve\_reference

该函数用于解析给定地址的字符串表示形式所对应的函数的地址。如果字符串是以“NewLibrary.NewFunctionName”的形式给出，它会首先尝试找到已加载的名为“NewLibrary.dll”的库并返回该库中的函数地址。如果未找到该库，则假定该字符串表示的函数是一个遗留 DLL 中的函数，然后在除给定库之外的所有已加载库中搜索该函数。如果找到该函数，则返回其地址。如果找不到该函数，则返回 NULL。

## get\_function\_address

一个函数 **get\_function\_address** 的实现，它根据传入的模块句柄 **hLibrary**，函数名称或者序号等信息来获取一个函数的地址。该函数首先解析模块的 PE 头部信息，然后遍历导出表中的符号，查找目标函数名称或者序号所对应的地址。如果找到了该地址，则返回该地址。如果地址指向另一个 DLL 中的函数，则调用 **resolve\_reference** 函数来解析该地址，返回其真正的函数地址。如果没有找到符合条件的地址，则返回 NULL。

该函数中用到了一些 Windows PE 文件格式相关的数据结构和宏定义，例如 **PIMAGE\_DOS\_HEADER**，**PIMAGE\_NT\_HEADERS**，**RVA** 等。同时还用到了一些辅助函数，例如 **resolve\_reference** 函数，它用于解析导出表中的函数地址指向其他 DLL 中的函数的情况。

## get\_library\_address

这是一个用于获取某个库的句柄（handle）的函数。它接受两个参数，一个是库的路径，另一个是一个布尔值，表示是否加载该库。

函数首先检查指定路径的库是否已经加载，如果是，它将返回该库的句柄。如果没有加载，则它将使用 **LdrLoadDll** 函数来加载该库，该函数是动态链接库管理器的一部分，用于从磁盘上的文件加载库。如果加载失败，该函数将返回 NULL。

# Entry

## Go

这是 C 代码中的一个函数，似乎用于获取 LSASS 进程的句柄并创建内存转储。 它接收LSASS的进程ID、是否将dump写入磁盘、是否fork LSASS、是否获得提升权限等几个参数。

该函数首先将几个变量初始化为其默认值，并使用 BeaconDataParse() 函数解析输入参数。 然后它会尝试启用调试权限并获取 LSASS 的进程 ID（如果未将其作为参数提供）。

如果设置了“get\_pid\_and\_leave”参数，该函数将简单地返回 LSASS 的进程 ID 并退出。

如果设置了“use\_silent\_process\_exit”参数，该函数将在给定路径创建一个文件夹，并让 Windows 错误报告进程为 LSASS 创建内存转储。

如果设置了“elevate\_handle”或“duplicate\_elevate”参数，该函数将检查当前用户是否为 SYSTEM，如果不是则模拟 SYSTEM。

然后该函数尝试使用 obtain\_lsass\_handle() 函数获取 LSASS 的句柄，并传入提供给该函数的几个参数。

如果设置了“use\_seclogon\_leak\_local”，该函数将简单地返回。

如果设置了“use\_lsass\_shtinkering”，该函数将调用 werfault\_shtinkering() 函数并退出。

最后，如果设置了“use\_valid\_sig”，该函数将设置签名。

如果函数中的任何操作失败，函数将跳转到“清理”标签并退出。

## Cleanup

这段代码是一个 C 语言程序的一部分，它包含了一些资源释放操作，用于在程序运行结束时清理一些被分配的内存和处理过程中创建的其他资源。代码包含一些条件语句和宏定义，这些可能是针对特定平台或编译选项的。其中的操作包括：

* 杀死 forked\_lsass 进程，如果它被创建了且处理过程中发生了错误；
* 关闭一个名为 hProcess 的句柄；
* 从内存中删除一个名为 dc 的结构体的指针所指向的内存段；
* 如果程序未成功执行并且已将转储文件写入磁盘，则删除该文件；
* 释放一个名为 hSnapshot 的资源；
* 杀死一些由程序创建的进程，然后释放一个名为 created\_processes 的指针所指向的内存段；
* 如果已经模拟了一个安全标识符，则恢复原来的安全标识符并关闭名为 hImpersonate 的句柄。

最后一个条件语句 "#elif defined(NANO) && defined(EXE)" 可能是用于在不同的编译选项下执行不同的代码块，但我们无法判断这个条件语句的含义，因为这只是代码的一部分而没有上下文。

## Usage

它输出命令行参数的使用方法和说明。它接受一个字符串参数**procname**，表示程序的名称。函数中包含了不同选项的说明，例如通过哪些方式获取LSASS句柄并创建转储文件、如何让WerFault.exe创建转储文件、如何避免以高权限打开LSASS等等。函数使用**PRINT**宏打印字符串，这个宏可能是自定义的，作用类似于标准库函数**printf**。

## NanoDumpSSP

作用是生成一个minidump文件，用于在程序崩溃时保存程序状态，以便后续分析和调试。该函数中的变量dump\_path表示生成的minidump文件的路径，use\_valid\_sig表示是否使用有效的minidump文件头签名。

代码首先创建一个dump\_context结构体，设置签名，然后通过NtCurrentProcess()获取当前进程句柄。接下来，代码通过allocate\_memory()函数分配内存来写入dump，最后将生成的dump写入文件并返回是否成功的布尔值bReturnValue。如果use\_valid\_sig为false，则可以通过encrypt\_dump()函数对生成的dump进行加密或混淆。如果文件无法写入，则会跳转到cleanup标签并返回false。

## NanoDumpPPL

创建Windows的Minidump文件的函数。Minidump文件是指向在Windows操作系统崩溃时包含进程状态信息的二进制文件。该函数接受多个参数，包括输出文件路径、进程ID、是否复制句柄等，然后将进程的状态信息写入到输出文件中。函数的实现过程中使用了一些系统API来获取所需信息，如GetCommandLineW()和CommandLineToArgvW()用于获取命令行参数，而obtain\_lsass\_handle()函数用于获取进程的句柄。函数执行成功后，会在控制台输出一条包含生成Minidump文件大小的调试信息。

# handle

# impersonate

# load\_ssp

# malseclogon

# modules

# nanodump

## writeat

这个函数将给定的数据写入到dump文件中的指定RVA（相对虚拟地址）处。参数包括：

* Pdump\_context dc: 表示dump文件上下文的结构体指针，包含了dump文件的基地址等信息。
* ULONG32 rva: 表示要写入数据的RVA，相对于dump文件的基地址。
* const PVOID data: 表示要写入的数据指针。
* unsigned size: 表示要写入的数据长度。

函数通过计算RVA相对于dump文件基地址的偏移量来获取目标内存地址，然后使用memcpy函数将数据写入目标地址。

## Append

该函数用于将数据追加到dump中。函数会计算数据追加后的RVA（相对虚拟地址），并检查是否超出32位地址空间或是否超出了dump的最大大小限制。如果没有超出限制，则将数据写入dump中并更新dump的RVA。函数返回一个BOOL值表示追加操作是否成功。

## write\_header

该函数用于在Dump文件中写入头信息。Dump文件包含一个文件头，其中包含一些元数据，如文件的版本，实现版本，时间戳等。MiniDumpHeader是用于表示Dump文件头的结构体。此函数使用提供的Dump上下文和MiniDumpHeader结构体填充头信息。它还将Header写入Dump文件。最后，该函数返回一个布尔值，指示写操作是否成功。如果操作成功，它将返回TRUE，否则将返回FALSE。

## write\_directory

这段代码定义了一个名为**write\_directory**的函数，该函数用于将一个**MiniDumpDirectory**结构体写入到dump文件中的目录表中。

函数接受两个参数：**dc**表示一个指向**dump\_context**结构体的指针，包含了dump文件的一些基本信息；**directory**表示要写入目录表中的一个**MiniDumpDirectory**结构体。

函数首先将**MiniDumpDirectory**结构体转换成字节数组，然后将字节数组追加到dump文件的末尾，这个过程使用了之前定义的**append**函数。如果写入成功，函数返回**TRUE**；否则返回**FALSE**。

## write\_directories

这段代码实现了写入三个MiniDumpDirectory结构，分别对应于三个不同的stream类型：SystemInfoStream，ModuleListStream和Memory64ListStream。每个MiniDumpDirectory结构包含了这个stream的大小（DataSize）和RVA（Rva），并且这些值都在之后计算并写入。

函数首先创建一个MiniDumpDirectory结构，将其StreamType字段设置为SystemInfoStream并将DataSize和Rva字段设置为0，然后调用write\_directory函数将其写入到dump文件中。然后，这个过程对应于其他两个stream类型，分别设置相应的MiniDumpDirectory结构和调用write\_directory函数。

如果任何一个write\_directory函数调用失败，则整个函数返回FALSE，否则返回TRUE。

## write\_system\_info\_stream

这似乎是生成系统信息流的 C 语言程序的一部分。 它定义了一个名为“write\_system\_info\_stream”的函数，该函数将指向“dump\_context”结构的指针作为参数。 在这个函数中，定义了一个“MiniDumpSystemInfo”结构并将其初始化为零。 然后使用进程环境块 (PEB) 检索系统信息，并将其存储在“MiniDumpSystemInfo”结构的适当字段中。

然后该函数创建一个字节数组，其中包含“MiniDumpSystemInfo”结构中的数据，并使用“附加”函数将其写入转储上下文。 流的大小和 RVA 也被写入 MiniDumpSystemInfo 目录中的转储上下文。

在没有更多上下文的情况下，不清楚转储上下文代表什么或这段代码是什么的一部分。

## write\_module\_list\_stream

实现了写入Windows进程的模块信息到一个minidump文件中的功能。在Windows操作系统中，minidump文件可以用于诊断应用程序的崩溃、故障和错误，包含了应用程序的堆栈、内存信息和模块信息等重要数据，有助于开发人员和系统管理员进行故障排除和问题分析。

具体来说，这段代码通过调用Windows API获取指定进程中一组重要的DLL模块信息，将这些模块的全路径名依次写入minidump文件中的ModuleListStream流中，并记录每个模块路径名的相对虚拟地址（RVA）。然后，该代码根据minidump文件格式的规定，将每个模块的重要信息（如基地址、大小、校验和、时间戳等）按照指定的顺序依次写入一个BYTE数组中，并将该BYTE数组写入minidump文件中的ModuleStream流中，以供后续的分析使用。最后，该代码将记录的模块数量写入ModuleListStream流中，并返回一个指向module\_info结构体链表的指针，该结构体记录了每个模块的详细信息。如果写入过程中出现错误，该代码会将该链表释放，并返回NULL。

## is\_important\_module

函数的作用是判断传入的地址是否在指定的模块列表中。函数接收两个参数，第一个参数为地址，第二个参数为模块列表。

该函数先将模块列表的头指针赋值给 **curr\_module** 变量，然后通过一个循环遍历整个模块列表。在每次循环中，函数会检查当前地址是否在当前模块的内存地址范围内，如果是，则返回 **TRUE**，表示该地址在指定的重要模块中；否则，将当前指针指向下一个模块，直到遍历完整个模块列表都没有找到该地址，最终返回 **FALSE**，表示该地址不在指定的重要模块中。

## get\_memory\_ranges

这段代码是用于获取需要在 MiniDump 文件中记录的内存区域范围。MiniDump 文件是一个 Windows 的调试信息文件，它用于记录在应用程序崩溃时的堆栈跟踪、内存状态和程序状态等信息，以便于开发人员调试和修复程序错误。

函数中首先调用了 NtQueryVirtualMemory 函数获取当前进程的虚拟内存信息。然后根据一些条件（如非空闲、非映射、无访问障碍等）筛选出需要记录的内存区域，并构造出 MiniDumpMemoryDescriptor64 结构体来保存这些内存区域的信息。最后，将这些结构体串成一个链表返回给调用者。

## write\_memory64\_list\_stream

这是一个 C 函数，它将内存转储写入 MiniDump 格式的文件。 该函数接受转储上下文和模块列表，并返回指向 MiniDumpMemoryDescriptor64 结构的指针。 该函数首先使用 get\_memory\_ranges 函数获取内存范围。 如果获取内存范围失败，则返回 NULL。 然后它将范围数写入文件，然后是实际内存内容的 RVA。 然后它将每个内存范围的开始和大小写入文件。 它将 Memory64ListStream 目录的长度和 RVA 写入文件。 最后，它将所有选定的内存范围转储到文件中。

该函数使用多个辅助函数将数据附加到文件、将数据写入文件中的特定位置、分配内存以及释放内存描述符的链表。 它还使用 NtReadVirtualMemory 函数从目标进程读取内存。

总体而言，此函数负责将 Memory64ListStream 写入 MiniDump 文件，其中包含有关目标进程的内存范围的信息。

## NanoDumpWriteDump

这是一个创建 NanoDump 文件的 C 函数。 该函数采用转储上下文并将标头、目录、系统信息流、模块列表流和 memory64 列表流写入文件。 它返回一个布尔值，指示操作是否成功。

该函数首先使用 write\_header 函数写入标头。 如果此操作失败，则函数返回 FALSE。

然后它使用 write\_directories 函数写入目录。 如果此操作失败，则函数返回 FALSE。

接下来，它使用 write\_system\_info\_stream 函数写入系统信息流。 如果此操作失败，则函数返回 FALSE。

然后该函数使用 write\_module\_list\_stream 函数写入模块列表流。 如果此操作失败，该函数将使用 free\_linked\_list 函数释放为模块列表分配的内存并返回 FALSE。

然后该函数使用 write\_memory64\_list\_stream 函数写入 memory64 列表流。 如果此操作失败，该函数将使用 free\_linked\_list 函数释放为模块列表和内存范围分配的内存并返回 FALSE。

最后，该函数使用 free\_linked\_list 函数释放分配给模块列表和内存范围的内存，并返回 TRUE，表示 NanoDump 文件创建成功。

# output

# restore\_signature

# shtinkering

# spoof\_callstack

# syscalls

# syscalls-asm.asm

# token\_priv

# utils

# werfault

cleanup

ppl

ppl\_utils