← Windows Process Injection: WordWarping, Hyphentension, AutoCourgette, Streamception, Oleum, ListPlanting, Treepoline

Shellcode: Using the Exception Directory to find GetProcAddress →

Shellcode: Loading .NET Assemblies From Memory

Posted on May 10, 2019

Introduction

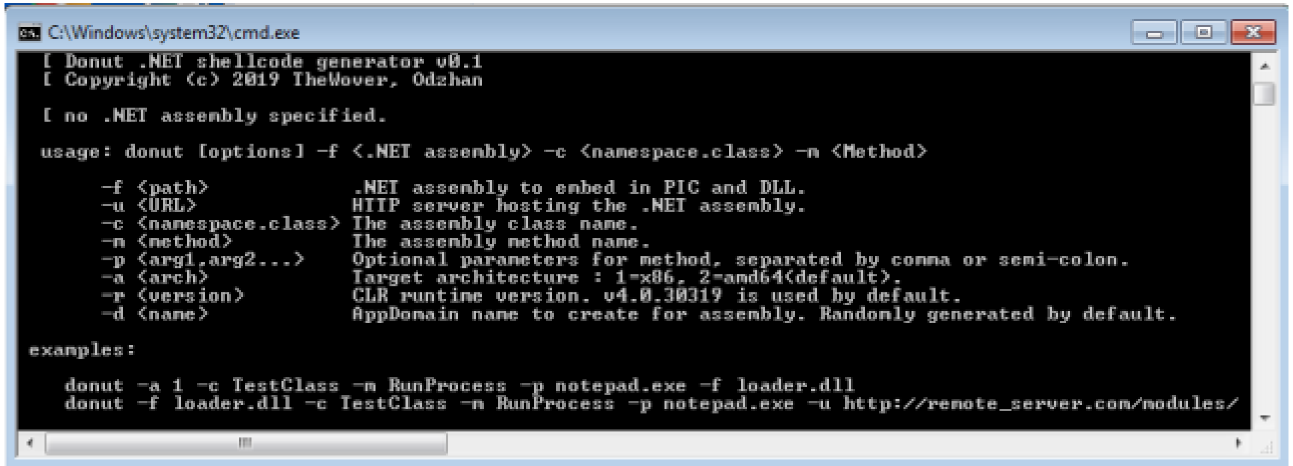
The dot net Framework can be found on almost every device running Microsoft Windows. It is popular among professionals involved in both attacking (Red Team) and defending (Blue Team) a Windows-based device. In 2015, the [Antimalware Scan Interface](#) (AMSI) was integrated with various Windows components used to execute scripts (VBScript,

Search

Recent Posts

- MiniDumpWriteDump via COM+ Services DLL
- Windows Process Injection: Asynchronous Procedure Call (APC)
- Windows Process Injection: KnownDlls Cache Poisoning
- Windows Process Injection: Tooltip or Common Controls
- Windows Process Injection: Breaking BaDDEr
- Windows Process Injection: DNS Client API

JScript, PowerShell). Around the same time, enhanced logging or Script Block Logging was added to PowerShell that allows capturing the full contents of scripts being executed, thereby defeating any obfuscation used. To remain ahead of blue teams, red teams had to go another layer deeper into the dot net framework by using assemblies. Typically written in C#, assemblies provide red teams with all the functionality of PowerShell, but with the distinct advantage of loading and executing entirely from memory. In this post, I will briefly discuss a tool called Donut, that when given a .NET assembly, class name, method, and optional parameters, will generate a position-independent code (PIC) or shellcode that can load a .NET assembly from memory. The project was a collaborative effort between myself and TheWover who has blogged about donut here.



Common Language Runtime (CLR) Hosting Interfaces

The CLR is the virtual machine component while the ICorRuntimeHost interface available since v1.0 of the framework (released in 2002) facilitates hosting .NET assemblies. This interface was superseded by ICLRRuntimeHost when v2.0 of the framework was released in 2006, and this was superseded by ICLRMetaHost when v4.0 of the framework was

- Windows Process Injection: Multiple Provider Router (MPR) DLL and Shell Notifications
- Windows Process Injection: Winsock Helper Functions (WSHX)
- Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL
- Shellcode: In-Memory Execution of DLL
- Windows Process Injection : Windows Notification Facility
- How Red Teams Bypass AMSI and WLDP for .NET Dynamic Code
- Windows Process Injection: KernelCallbackTable used by FinFisher / FinSpy
- Windows Process Injection: CLIPBRDWNDCLASS
- Shellcode: Using the Exception Directory to find GetProcAddress
- Shellcode: Loading .NET Assemblies From Memory
- Windows Process Injection: WordWarping, Hyphentension, AutoCourgette, Streamception, Oleum, ListPlanting, Treepoline
- Shellcode: A reverse shell for Linux in C with support for TLS/SSL
- Windows Process Injection: Print Spooler
- How the L0pht (probably) optimized attack against the LanMan hash.

released in 2009. Although deprecated, `ICorRuntimeHost` currently provides the easiest way to load assemblies from memory. There are a variety of ways to instantiate this interface, but the most popular appears to be through one of the following:

- CoInitializeEx and CoCreateInstance
- CorBindToRuntime or CorBindToRuntimeEx
- CLRCreateInstance and ICLRRuntimeInfo

`CorBindToRuntime` and `CorBindToRuntimeEx` functions perform the same operation, but the `CorBindToRuntimeEx` function allows us to specify the behavior of the CLR. `CLRCreateInstance` avoids having to initialize Component Object Model (COM) but is not implemented prior to v4.0 of the framework. The following code in C++ demonstrates running a dot net assembly from memory.

```
#include <windows.h>
#include <oleauto.h>
#include <mscoree.h>
#include <comdef.h>

#include <cstdio>
#include <cstdint>
#include <cstring>
#include <cstdlib>
#include <sys/stat.h>

#import "mscorlib.tlb" raw_interfaces_only

void rundotnet(void *code, size_t len) {
    HRESULT                  hr;
```

```
ICorRuntimeHost          *icrh;
IUnknownPtr              iu;
mscorlib::_AppDomainPtr  ad;
mscorlib::_AssemblyPtr   as;
mscorlib::_MethodInfoPtr mi;
VARIANT                  v1, v2;
SAFEARRAY                *sa;
SAFEARRAYBOUND           sab;

printf("CoCreateInstance(ICorRuntimeHost).\n");
hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);

hr = CoCreateInstance(
  CLSID_CorRuntimeHost,
  NULL,
  CLSCTX_ALL,
  IID_ICorRuntimeHost,
  (LPVOID*)&icrh);

if(FAILED(hr)) return;

printf("ICorRuntimeHost::Start()\n");
hr = icrh->Start();
if(SUCCEEDED(hr)) {
  printf("ICorRuntimeHost::GetDefaultDomain()\n");
  hr = icrh->GetDefaultDomain(&iu);
  if(SUCCEEDED(hr)) {
    printf("IUnknown::QueryInterface()\n");
    hr = iu->QueryInterface(IID_PPV_ARGS(&ad));
    if(SUCCEEDED(hr)) {
      sab.lLbound   = 0;
      sab.cElements = len;
```


- Shellcode: x86 optimizations part 1
- WanaCryptor File Encryption and Decryption
- Shellcode: Dual Mode (x86 + amd64) Linux shellcode
- Shellcode: Fido and how it resolves GetProcAddress and LoadLibraryA
- Shellcode: Dual mode PIC for x86 (Reverse and Bind Shells for Windows)
- Shellcode: Solaris x86
- Shellcode: Mac OSX amd64
- Shellcode: Resolving API addresses in memory
- Shellcode: A Windows PIC using RSA-2048 key exchange, AES-256, SHA-3
- Shellcode: Execute command for x32/x64 Linux / Windows / BSD
- Shellcode: Detection between Windows/Linux/BSD on x86 architecture
- Shellcode: FreeBSD / OpenBSD amd64
- Shellcode: Linux amd64
- Shellcodes: Executing Windows and Linux Shellcodes
- DLL/PIC Injection on Windows from Wow64 process
- Asmcodes: Platform Independent PIC for Loading DLL and Executing Commands

```
                printf("SafeArrayCreate()\n");
                sa = SafeArrayCreate(VT_UI1, 1, &sab);
                if(sa != NULL) {
                  CopyMemory(sa->pvData, code, len);
                  printf("AppDomain::Load_3()\n");
                  hr = ad->Load_3(sa, &as);
                  if(SUCCEEDED(hr)) {
                    printf("Assembly::get_EntryPoint()\n");
                    hr = as->get_EntryPoint(&mi);
                    if(SUCCEEDED(hr)) {
                      v1.vt    = VT_NULL;
                      v1.plVal = NULL;
                      printf("MethodInfo::Invoke_3()\n");
                      hr = mi->Invoke_3(v1, NULL, &v2);
                      mi->Release();
                    }
                    as->Release();
                  }
                  SafeArrayDestroy(sa);
                }
                ad->Release();
              }
              iu->Release();
            }
            icrh->Stop();
          }
          icrh->Release();
      }

      int main(int argc, char *argv[])
      {
          void *mem;
```

```c
    struct stat fs;
    FILE *fd;

    if(argc != 2) {
      printf("usage: rundotnet <.NET assembly>\n");
      return 0;
    }

    // 1. get the size of file
    stat(argv[1], &fs);

    if(fs.st_size == 0) {
      printf("file is empty.\n");
      return 0;
    }

    // 2. try open assembly
    fd = fopen(argv[1], "rb");
    if(fd == NULL) {
      printf("unable to open \"%s\".\n", argv[1]);
      return 0;
    }
    // 3. allocate memory
    mem = malloc(fs.st_size);
    if(mem != NULL) {
      // 4. read file into memory
      fread(mem, 1, fs.st_size, fd);
      // 5. run the program from memory
      rundotnet(mem, fs.st_size);
      // 6. free memory
      free(mem);
    }
```

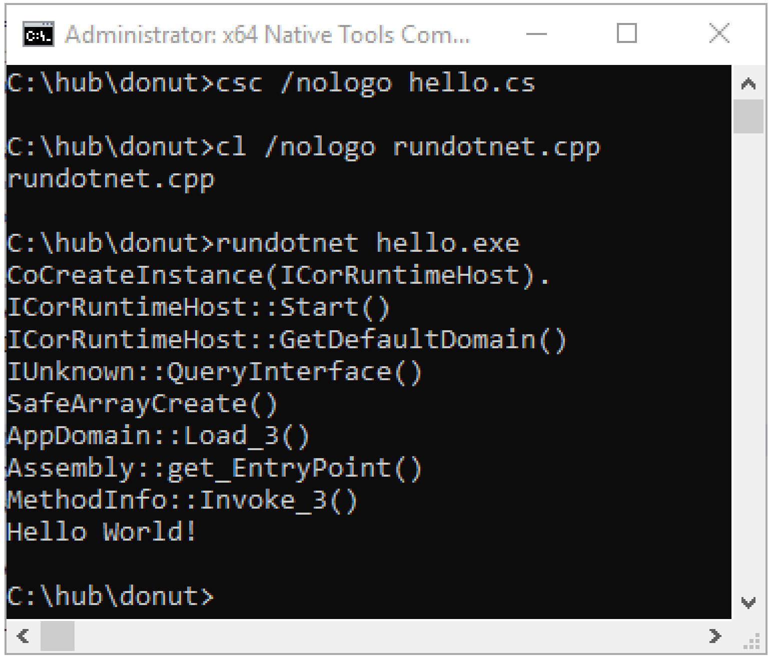
```
        // 7. close assembly
        fclose(fd);

        return 0;
    }
```

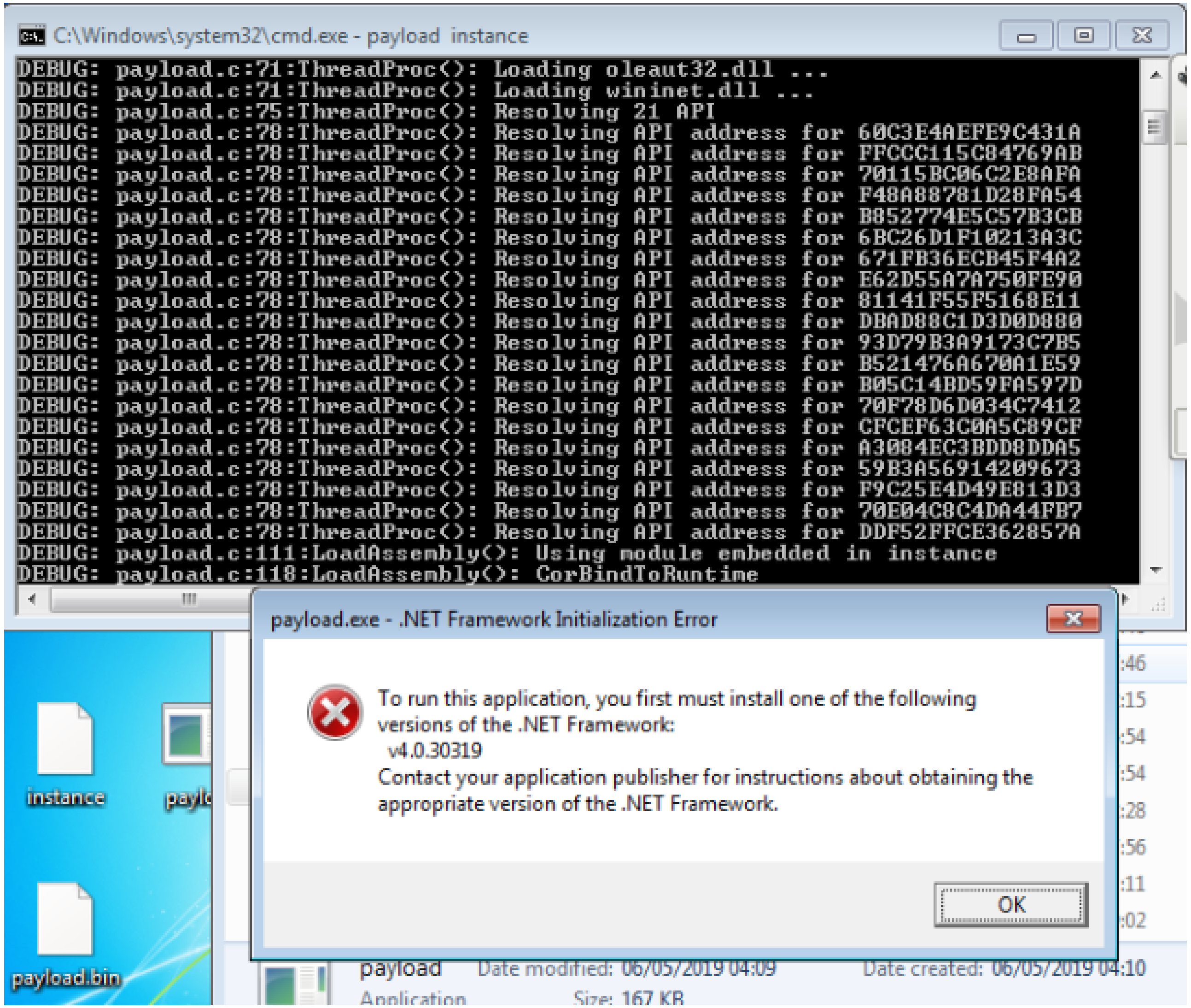
The following is a simple Hello, World! example in C# that when compiled with csc.exe will generate a dot net assembly for testing the loader.

```
    // A Hello World! program in C#.
    using System;
    namespace HelloWorld
    {
        class Hello
        {
            static void Main()
            {
                Console.WriteLine("Hello World!");
            }
        }
    }
```

Compiling and running both of these sources gives the following results.



That's a basic implementation of executing dot net assemblies and doesn't take into consideration what runtime versions of the framework are supported. The shellcode works differently by resolving the address of `CorBindToRuntime` and `CLRCreateInstance` together which is similar to AssemblyLoader by subTee. If `CLRCreateInstance` is successfully resolved and invocation returns `E_NOTIMPL` or "Not implemented", we execute `CorBindToRuntime` with the `pwszVersion` parameter set to NULL, which simply requests the latest version available. If we request a specific version from `CorBindToRuntime` that is not supported by the system, a host process running the shellcode might display an error message. For example, the following screenshot shows a request for v4.0.30319 on a Windows 7 machine that only supports v3.5.30729.5420.



You may be asking why the OLE functions used in the hosting example are not also used in the shellcode. OLE functions are sometimes referenced in another DLL like COMBASE instead of OLE32. xGetProcAddress can handle forward references, but for now at least, the shellcode uses a combination of `CorBindToRuntime` and `CLRCreateInstance`. `CoCreateInstance` may be used in newer versions.

Defining .NET Types

Types are accessible from an unmanaged C++ application using the `#import` directive. The hosting example uses `_AppDomain`, `_Assembly` and `_MethodInfo` interfaces defined in `mscorlib.tlb`. The problem, however, is that there's no definition of the interfaces anywhere in the public version of the Windows SDK. To use a dot net type from lower-level languages like assembly or C, we first have to manually define them. The type information can be enumerated using the LoadTypeLib API which returns a pointer to the ITypeLib interface. This interface will retrieve information about the library while ITypeInfo will retrieve information about the library interfaces, methods and variables. I found the open source application Olewoo useful for examining `mscorlib.tlb`. If we ignore all the concepts of Object Oriented Programming (OOP) like class, object, inheritance, encapsulation, abstraction, polymorphism..etc, an interface can be viewed from a lower-level as nothing more than a pointer to a data structure containing pointers to functions/methods. I could not find any definition of the required interfaces online except for one file in phplib that partially defines the `_AppDomain` interface. Based on that example, I created the other interfaces necessary for loading assemblies. The following method is a member of the `_AppDomain` interface.

```
HRESULT (STDMETHODCALLTYPE *InvokeMember_3)(
  IType        *This,
  BSTR         name,
  BindingFlags invokeAttr,
  IBinder      *Binder,
```

```
          VARIANT       Target,
          SAFEARRAY     *args,
          VARIANT       *pRetVal);
```

Although no methods of the `IBinder` interface are used in the shellcode and the type could safely be changed to `void *`, the following is defined for future reference. The `DUMMY_METHOD` macro simply defines a function pointer.

```
typedef struct _Binder IBinder;

#undef DUMMY_METHOD
#define DUMMY_METHOD(x) HRESULT ( STDMETHODCALLTYPE *dummy_##x )(

typedef struct _BinderVtbl {
    HRESULT ( STDMETHODCALLTYPE *QueryInterface )(
      IBinder * This,
      /* [in] */ REFIID riid,
      /* [iid_is][out] */ void **ppvObject);

    ULONG ( STDMETHODCALLTYPE *AddRef )(
      IBinder * This);

    ULONG ( STDMETHODCALLTYPE *Release )(
      IBinder * This);

    DUMMY_METHOD(GetTypeInfoCount);
    DUMMY_METHOD(GetTypeInfo);
    DUMMY_METHOD(GetIDsOfNames);
    DUMMY_METHOD(Invoke);
```

```
    DUMMY_METHOD(ToString);
    DUMMY_METHOD(Equals);
    DUMMY_METHOD(GetHashCode);
    DUMMY_METHOD(GetType);
    DUMMY_METHOD(BindToMethod);
    DUMMY_METHOD(BindToField);
    DUMMY_METHOD(SelectMethod);
    DUMMY_METHOD(SelectProperty);
    DUMMY_METHOD(ChangeType);
    DUMMY_METHOD(ReorderArgumentArray);
} BinderVtbl;

typedef struct _Binder {
  BinderVtbl *lpVtbl;
} Binder;
```

Methods required to load assemblies from memory are defined in payload.h.

Donut Instance

The shellcode will always be combined with a block of data referred to as an *Instance*. This can be considered the "data segment" of the shellcode. It contains the names of DLL to load before attempting to resolve API, 64-bit hashes of API strings, COM GUIDs relevant for loading .NET assemblies into memory and decryption keys for both the *Instance*, and the *Module* if one is stored on a staging server. Many shellcodes written in C tend to store strings on the stack, but tools like FireEye Labs Obfuscated String Solver can recover them with relative ease, helping to analyze the code much faster. One advantage of keeping strings in a separate data block is when it comes to the permutation of the code. It's possible to change the code while retaining the functionality, but never having to work with "read-only" immediate values that would complicate the process and significantly increase the size of the code. The following structure represents what is placed after a `call` opcode and before a `pop ecx / pop rcx`. The `fastcall` convention is used for both x86 and x86-64 shellcodes and this makes it convenient to load a pointer to the *Instance* in `ecx` or `rcx` register.

```
typedef struct _DONUT_INSTANCE {
    uint32_t    len;                          // total size of instar
    DONUT_CRYPT key;                          // decrypts instance
    // everything from here is encrypted

    int         dll_cnt;                      // the number of DLL tc
    char        dll_name[DONUT_MAX_DLL][32];  // a list of DLL string
    uint64_t    iv;                           // the 64-bit initial \
    int         api_cnt;                      // the 64-bit hashes oi

    union {
      uint64_t  hash[48];                     // holds up to 48 api h
      void     *addr[48];                     // holds up to 48 api a
      // include prototypes only if header included from payload.h
      #ifdef PAYLOAD_H
      struct {
        // imports from kernel32.dll
        LoadLibraryA_t             LoadLibraryA;
        GetProcAddress_t           GetProcAddress;
        VirtualAlloc_t             VirtualAlloc;
        VirtualFree_t              VirtualFree;

        // imports from oleaut32.dll
```

```
        SafeArrayCreate_t           SafeArrayCreate;
        SafeArrayCreateVector_t     SafeArrayCreateVector;
        SafeArrayPutElement_t       SafeArrayPutElement;
        SafeArrayDestroy_t          SafeArrayDestroy;
        SysAllocString_t            SysAllocString;
        SysFreeString_t             SysFreeString;

        // imports from wininet.dll
        InternetCrackUrl_t          InternetCrackUrl;
        InternetOpen_t              InternetOpen;
        InternetConnect_t           InternetConnect;
        InternetSetOption_t         InternetSetOption;
        InternetReadFile_t          InternetReadFile;
        InternetCloseHandle_t       InternetCloseHandle;
        HttpOpenRequest_t           HttpOpenRequest;
        HttpSendRequest_t           HttpSendRequest;
        HttpQueryInfo_t             HttpQueryInfo;

        // imports from mscoree.dll
        CorBindToRuntime_t          CorBindToRuntime;
        CLRCreateInstance_t         CLRCreateInstance;
      };
      #endif
    } api;

    // GUID required to load .NET assembly
    GUID xCLSID_CLRMetaHost;
    GUID xIID_ICLRMetaHost;
    GUID xIID_ICLRRuntimeInfo;
    GUID xCLSID_CorRuntimeHost;
    GUID xIID_ICorRuntimeHost;
    GUID xIID_AppDomain;
```

```
    DONUT_INSTANCE_TYPE type;  // PIC or URL

    struct {
      char url[DONUT_MAX_URL];
      char req[16];            // just a buffer for "GET"
    } http;

    uint8_t     sig[DONUT_MAX_NAME];          // string to hash
    uint64_t    mac;                          // to verify decryptior

    DONUT_CRYPT mod_key;       // used to decrypt module
    uint64_t    mod_len;       // total size of module

    union {
      PDONUT_MODULE p;         // for URL
      DONUT_MODULE  x;         // for PIC
    } module;
} DONUT_INSTANCE, *PDONUT_INSTANCE;
```

Donut Module

A dot net assembly is stored in a data structure referred to as a *Module*. It can be stored with an *Instance* or on a staging server that the shellcode will retrieve it from. Inside the module will be the assembly, class name, method, and optional parameters. The `sig` value will contain a random 8-byte string that when processed with the *Maru* hash function will generate a 64-bit value that should equal the value of `mac`. This is to verify decryption of the module was successful. The *Module* key is stored in the *Instance* embedded with the shellcode.

```
// everything required for a module goes into the following structure
typedef struct _DONUT_MODULE {
    DWORD   type;                                   // EXE or DLL
    WCHAR   runtime[DONUT_MAX_NAME];                // runtime versic
    WCHAR   domain[DONUT_MAX_NAME];                 // domain name tc
    WCHAR   cls[DONUT_MAX_NAME];                    // name of class
    WCHAR   method[DONUT_MAX_NAME];                 // name of methoc
    DWORD   param_cnt;                              // number of para
    WCHAR   param[DONUT_MAX_PARAM][DONUT_MAX_NAME]; // string paramet
    CHAR    sig[DONUT_MAX_NAME];                    // random string
    ULONG64 mac;                                    // to verify deci
    DWORD   len;                                    // size of .NET a
    BYTE    data[4];                                // .NET assembly
} DONUT_MODULE, *PDONUT_MODULE;
```

Random Keys

On Windows, CryptGenRandom generates cryptographically secure random values while on Linux, `/dev/urandom` is used instead of `/dev/random` because the latter blocks on read attempts. Thomas Huhn writes in Myths about /dev/urandom that `/dev/urandom` is the preferred source of cryptographic randomness on Linux. Now, I don't suggest any of you reuse CreateRandom to generate random keys, but that's how they're generated in Donut.

Random Strings

Application Domain names are generated using a random string unless specified by the user generating a payload. If a donut module is stored on a staging server, a random name is generated for that too. The function that handles this is aptly named GenRandomString. Using random bytes from `CreateRandom`, a string is derived from the letters "HMN34P67R9TWCXYF". The selection of these letters is based on a post by trepidacious about unambiguous characters.

Symmetric Encryption

An involution is simply a function that is its own inverse and many tools use involutions to obfuscate the code. If you've ever reverse engineered malware, you will no doubt be familiar with the eXclusive-OR operation that is used quite a lot because of its simplicity. A more complicated example of involutions can be the non-linear operation used for the Noekeon block cipher. Instead of involutions, Donut uses the Chaskey block cipher in Counter (CTR) mode to encrypt the module with the decryption key embedded in the shellcode. If a Donut module is recovered from a staging server, the only way to get information about what's inside it is to recover the shellcode, find a weakness with the `CreateRandom` function or break the Chaskey cipher.

```
static void chaskey(void *mk, void *p) {
    uint32_t i,*w=p,*k=mk;

    // add 128-bit master key
    for(i=0;i<4;i++) w[i]^=k[i];

    // apply 16 rounds of permutation
    for(i=0;i<16;i++) {
      w[0] += w[1],
      w[1]  = ROTR32(w[1], 27) ^ w[0],
      w[2] += w[3],
```

```
        w[3]  = ROTR32(w[3], 24) ^ w[2],
        w[2] += w[1],
        w[0]  = ROTR32(w[0], 16) + w[3],
        w[3]  = ROTR32(w[3], 19) ^ w[0],
        w[1]  = ROTR32(w[1], 25) ^ w[2],
        w[2]  = ROTR32(w[2], 16);
      }
      // add 128-bit master key
      for(i=0;i<4;i++) w[i]^=k[i];
    }
```

Chaskey was selected because it's compact, simple to implement and doesn't contain constants that would be useful in generating simple detection signatures. The main downside is that Chaskey is relatively unknown and therefore hasn't received as much cryptanalysis as AES has. When Chaskey was first published in 2014, the recommended number of rounds was 8. In 2015, an [attack against 7](#) of the 8 rounds was discovered showing that the number of rounds was too low of a security margin. In response to this attack, the designers proposed 12 rounds, but Donut uses the Long-term Support (LTS) version with 16 rounds.

API Hashing

If the hash of an API string is well known in advance of a memory scan, detecting Donut would be much easier. It was suggested in [Windows API hashing with block ciphers](#) that introducing entropy into the hashing process would help code evade detection for longer. Donut uses the *Maru* hash function which is built atop of the Speck block cipher. It uses a Davies-Meyer construction and padding similar to what's used in MD4 and MD5. A 64-bit

Initial Value (IV) is generated randomly and used as the plaintext to encrypt while the API string is used as the key.

```
static uint64_t speck(void *mk, uint64_t p) {
    uint32_t k[4], i, t;
    union {
      uint32_t w[2];
      uint64_t q;
    } x;

    // copy 64-bit plaintext to local buffer
    x.q = p;

    // copy 128-bit master key to local buffer
    for(i=0;i<4;i++) k[i]=((uint32_t*)mk)[i];

    for(i=0;i<27;i++) {
      // donut_encrypt 64-bit plaintext
      x.w[0] = (ROTR32(x.w[0], 8) + x.w[1]) ^ k[0];
      x.w[1] =  ROTR32(x.w[1],29) ^ x.w[0];

      // create next 32-bit subkey
      t = k[3];
      k[3] = (ROTR32(k[1], 8) + k[0]) ^ i;
      k[0] =  ROTR32(k[0],29) ^ k[3];
      k[1] = k[2]; k[2] = t;
    }
    // return 64-bit ciphertext
    return x.q;
}
```

Summary

Donut is provided as a demonstration of CLR Injection through shellcode in order to provide red teamers a way to emulate adversaries and defenders a frame of reference for building analytics and mitigations. This inevitably runs the risk of malware authors and threat actors misusing it. However, we believe that the net benefit outweighs the risk. Hopefully, that is correct. Source code can be [found here].

Share this:

Twitter Facebook

Like 

One blogger likes this.

Related

Shellcode: In-Memory Execution of JavaScript, VBScript, JScript and XSL
In "assembly"

How Red Teams Bypass AMSI and WLDP for .NET Dynamic Code
In "assembly"

Shellcode: A Windows PIC using RSA-2048 key exchange, AES-256, SHA-3
In "assembly"

This entry was posted in assembly, encryption, malware, programming, security, shellcode, windows and tagged .net, c++, donut, dotnet, jscript, powershell, vbscript. Bookmark the permalink.

← Windows Process Injection: WordWarping, Hyphentension, AutoCourgette, Streamception, Oleum, ListPlanting, Treepoline

Shellcode: Using the Exception Directory to find GetProcAddress →

Leave a Reply

Enter your comment here...

modexp

Blog at WordPress.com.