SYSDREAM

IT Security Services

Nouvelle technique d'injection de code

Damien CAUQUIL

<d.cauquil@sysdream.com>

L'injection de code dans un processus est une technique connue depuis plusieurs années, et largement utilisée par de nombreux malwares, et de fait contrée par de nombreux logiciels anti-malware. Il existe cependant un moyen de contourner ces détections, tout en assurant l'injection d'une dll dans l'espace mémoire d'un autre processus, et cela de manière furtive, à l'insu de ces logiciels anti-malware.

Introduction

Les techniques d'injection de code sont nombreuses, mais la plus connue (et la plus usitée) est certainement celle employant le couple VirtualAllocEx/CreateRemoteThread. Cette technique est très facile à implémenter, et repose sur le fait que Windows ne filtre pas nativement les accès fait aux différents processus, ce qui permet l'allocation d'une zone mémoire dans le processus cible par le processus attaquant, et la création d'un thread dans ce même processus cible, thread exécutant un code malveillant. De nombreux sites traitent de cette technique, comme [1] et [2].

Cette technique d'injection est désormais caduque, car de nombreux logiciels anti-malwares vérifient l'emploi de ces deux APIs, afin de détecter une possible injection. De même, on a vu apparaître plusieurs moteurs de détection par heuristique, tels ceux implémentés dans A-Squared (EmsiSoft) et Kaspersky Anti-virus (Kaspersky).

Cependant, malgré tous ces contrôles, il reste un moyen très simple de faire exécuter du code par un processus: les Debug APIs de Windows. Ces APIs permettent de déboguer un processus, de récupérer des informations sur son statut, et d'accéder à sa mémoire. Tout ce dont nous avons besoin pour une injection.

I - Protections vs Debug APIs

Microsoft met à notre disposition un ensemble de routines destinées à déboguer des programmes, mais bien sûr les logiciels anti-malwares filtrent ces routines. Cependant, de nombreuses protections utilisent les Debug APIs, comme Armadillo, afin de protéger des binaires du reverse-engineering. Dans le cas d'Armadillo, l'utilisation des nanomites exploite cette possibilité de débogage, et la protection doit manipuler le contexte du processus débogué afin d'assurer son fonctionnement (je ne détaillerai pas ici le fonctionnement des nanomites, pour plus de détails reportez-vous à [3]).

Cette utilisation des Debug APIs par des protections ne doit pas être considérée par un anti-malware comme une attaque, bien que la frontière soit ténue. C'est pourquoi le moteur heuristique de ces anti-malwares cible particulièrement plusieurs APIs qui permettent de modifier le contexte d'un processus, telles que SetThreadContext et WriteProcessMemory, la première permettant de modifier le contexte du processus, et la seconde la mémoire de celui-ci. On peut donc supposer sans trop se tromper que ces deux fonctions sont hautement surveillées, la première peut-être moins à cause des protections que j'ai cité un peu plus

haut, car elle est utile dans le cadre d'une redirection de flux (modification du registre d'instruction, utilisée par certaines protections).

Notre objectif va donc être d'exploiter cette faiblesse de SetThreadContext, et plus généralement le laxisme de protection autour des Debug APIs, afin d'injecter dans le processus ciblé un code étranger.

Problèmes et solutions

A – redirection du flux d'exécution

SetThreadContext ne sera probablement autorisée qu'à modifier certains registres spécifiques, dits de "contrôle" (registre d'instruction EIP, registres de pile ESP/EBP, registre Eflags), elle peut donc agir sur ces quatre registres sans alerter les logiciels anti-malwares. Le registre EFlags est un registre spécifique dans l'architecture Intel x86, car il permet de modifier le comportement du processeur, et notamment d'activer le mode Single-Step.

Ce mode est particulier dans le sens où il va générer une exception Single-Step avant chaque exécution d'instruction, ce qui permet de faire ce qu'on appelle du mode "pas à pas". Ce mode peut être très intéressant: en effet, si on arrive à rediriger le flux d'exécution sur une instruction connue, on peut grâce au mode Single-Step exécuter cette instruction seule, pour ensuite rediriger l'exécution sur une autre instruction. De cette manière, on peut piloter le processus débogué afin de lui faire exécuter plusieurs instructions présentes en mémoire.

B – Allocation de mémoire

L'astuce précédente va nous permettre de faire exécuter du code par le processus cible, et on peut donc envisager de le faire appeler VirtualAlloc, qui sera exécutée sans problème car appelée par le processus cible. Dans le cas de mon implémentation, j'ai opté pour une solution alternative. Le but de l'outil est de pouvoir injecter n'importe quel code compilé dans un processus, code contenu dans une dll spécialement conçue, l'injecteur emploiera donc un procédé légèrement différent et un peu plus complexe..

Pour cela, l'injecteur va créer ce qu'on appelle un FileMapping, qui est un objet nommé accessible par plusieurs processus, dans lequel il insèrera le code de la dll. Ce code pourra être ensuite mappé dans le contexte du processus (par un appel à OpenFileMapping et MapViewOfFile), puis utilisé par le processus cible, utilisation que l'on forcera grâce à l'astuce présentée auparavant.

Pourquoi préférer l'emploi d'un FileMapping plutôt que d'un VirtualAlloc ? D'une part parce que l'appel à VirtualAlloc peut mettre la puce à l'oreille du logiciel anti-malware, et d'autre part parce que l'on peut garder une synchronisation entre le processus injecteur et le processus qui subit l'injection (à des fins de surveillance).

II - Procédé d'injection

Dans un premier temps, le processus injecteur va s'attacher au processus cible (en tant que debogueur) via l'API DebugActiveProcess. Il va ensuite être notifié par le système d'une exception CREATE_PROCESS_DEBUG_EVENT, qui permettra de récupérer un handle sur le processus cible sans passer par OpenProcess (fonction qui elle aussi est généralement

filtrée). Il ne reste plus qu'à activer le mode Single-Step, et à "piloter" le processus cible afin qu'il exécute le code désiré.

Pour des fins pratiques, l'injecteur va simplement modifier le programme afin qu'il lance tout seul le code injecté (sans passer par la méthode de redirection de flux vue précédemment), car un anti-malware peut prendre en compte le fait que le processus cible est en cours de débogage. L'injecteur va trouver une adresse située dans la mémoire du processus, y injecter un bout de code exécutable qui sera écrit non pas à l'aide de WriteProcessMemory, mais à l'aide d'une combinaison d'instructions assembleur appelées grâce à la méthode de redirection de flux qui revient à émuler WriteProcessMemory), puis écrire dans le FileMapping le code à injecter suivi d'un code rétablissant le code modifié de manière à ne laisser aucune trace. En relançant l'exécution, le processus injecteur va faire exécuter par le processus cible son code ajouté, qui mappera le FileMapping dans le contexte du processus, lancera le code injecté, puis rétablira le code modifié afin de ne laisser aucune trace de l'injection. De cette manière, l'injection est totalement furtive et ne laisse aucune trace, et de plus le code injecté est exécuté une fois le débogueur détaché, ce qui laisse croire aux protections anti-malware que c'est bien le processus cible seul qui a exécuté du code.

A – Emulation de fonction

Une des pierres angulaire de cette technique d'injection est l'émulation de la routine WriteProcessMemory, cette émulation se faisant à l'aide d'un ensemble d'instructions trouvées dans l'espace mémoire du processus cible. Afin de s'assurer d'avoir des adresses correspondant entre les deux processus (injecteur et cible), l'injecteur va chercher dans la bibliothèque de fonctions Kernel32.dll les octets correspondant aux codes-opérations des instructions utiles :

```
	// PUSH EBP
	dwPushEbp = FindInstB(0x55);
	// MOV BYTE PTR [EAX], BL
	dwMovBlToEax = FindInstW(0x1888);
	// PUSHAD
	dwPushad = FindInstB(0x60);
	// POPAD
	dwPopad = FindInstB(0x61);
```

Les routines FindInstB et FindInstW cherchent ces instructions dans Kernel32.dll.
Il peut ensuite émuler le fonctionnement de WriteProcessMemory :

```
void WriteByte(THREAD_PARAM *tp,void *address, unsigned char byte)
{
	CONTEXT saved,regs,test;
	regs.ContextFlags = CONTEXT_CONTROL;

	// on sauve le context actuel
	GetThreadContext(tp->hThread, &regs);
	saved = regs;

	// PUSH ADDRESS
	GetThreadContext(tp->hThread, &regs);
	regs.Ebp = (DWORD)address;
	regs.Eip = dwPushEbp;
	regs.EFlags |= 0x100;
	SetThreadContext(tp->hThread, &regs);
	PeekEvent(pEvents);
```

```
        WaitForEvent(pEvents);


        // POP EAX
        GetThreadContext(tp->hThread, &regs);
        regs.Eip = dwPopEax;
        regs.EFlags |= 0x100;
        SetThreadContext(tp->hThread, &regs);
        PeekEvent(pEvents);
        WaitForEvent(pEvents);

        // PUSH VAL
        GetThreadContext(tp->hThread, &regs);
        regs.Ebp = (DWORD)byte;
        regs.Eip = dwPushEbp;
        regs.EFlags |= 0x100;
        SetThreadContext(tp->hThread, &regs);
        PeekEvent(pEvents);
        WaitForEvent(pEvents);

        // POP EBX
        GetThreadContext(tp->hThread, &regs);
        regs.Eip = dwPopEbx;
        regs.EFlags |= 0x100;
        SetThreadContext(tp->hThread, &regs);
        PeekEvent(pEvents);
        WaitForEvent(pEvents);

        // MOV BYTE [EAX], BL
        GetThreadContext(tp->hThread, &regs);
        regs.Eip = dwMovBlToEax;
        regs.EFlags |= 0x100;
        SetThreadContext(tp->hThread, &regs);
        PeekEvent(pEvents);
        WaitForEvent(pEvents);

        SetThreadContext(tp->hThread,&saved);

}
```

Remarquez l'emploi de l'instruction "push ebp" pour fixer la valeur d'un registre tel que EBX ou EAX en utilisant seulement le contexte de contrôle :

```
        // PUSH VAL
        GetThreadContext(tp->hThread, &regs);
        regs.Ebp = (DWORD)byte;
        regs.Eip = dwPushEbp;
        regs.EFlags |= 0x100; // single step mode
        SetThreadContext(tp->hThread, &regs);
        // on relance l'exécution
        PeekEvent(pEvents);
        // on attend une exception SingleStep
        WaitForEvent(pEvents);

        // POP EBX
        GetThreadContext(tp->hThread, &regs);
        regs.Eip = dwPopEbx;
        regs.EFlags |= 0x100; // single step mode
        SetThreadContext(tp->hThread, &regs);
        PeekEvent(pEvents);
        WaitForEvent(pEvents);
```

En copiant de cette manière octet par octet le code à écrire, on arrive à la même finalité que WriteProcessMemory. De plus, vu que l'on sauvegarde le contexte de contrôle et qu'on le restaure en fin d'opération, il n'y a aucune répercussion sur l'exécution du programme.

B – Insertion du "bootstrap"

Le "bootstrap" est un bout de code qui sera chargé de mapper le FileMapping dans le contexte du processus cible, puis de lancer le code injecté. Ce bout de code est réalisé en assembleur x86, et intégré sous forme de code hexa dans l'injecteur. Il sera ensuite copié avec un appel à la fonction émulée WriteProcessMemory, à un endroit stratégique. Cet endroit stratégique pose problème : en effet, le code injecté doit être exécuté le plus rapidement possible, et nous devons donc nous assurer de l'injecter d'une part dans le contexte du processus (c'est-à-dire dans la section de code de celui-ci), et d'autre part le plus près possible de l'endroit où le processus est en pause.

Afin de réaliser ceci l'injecteur doit, une fois qu'il est attaché en tant que débogueur au processus cible, le mettre en pause et agir en conséquence. Lorsque le débogueur est attaché, le processus cible est suspendu par le système, mais rien ne dit que l'adresse de la prochaine exécution à exécuter (l'adresse qui permettrait d'agir au plus vite en y injectant le bootstrap) ne soit dans la section de code du processus cible. Il faut donc prévoir deux cas :

- Le cas où le processus cible est mis en pause lorsqu'il exécute du code dans la section de code
- Le cas où le processus cible est mis en pause lorsqu'il exécute une API, auquel cas il ne se trouve pas dans la section de code

Le second cas n'est pas si problématique, car il est assez aisé de scanner la pile à la recherche d'une adresse de retour tombant dans la section de code du processus cible. Cette identification de l'adresse d'injection est faite par ce petit bout de code (qui inclut un mini-désassembleur) :

```
do
{
      ReadProcessMemory(args->hProcess,(void *)dwEsp,(void
*)&dwRetAddr,sizeof(DWORD),&nbr);
      if ((dwRetAddr>=(DWORD)args->ImageBase) && (dwRetAddr <((DWORD)args-
>ImageBase+nt_stub.OptionalHeader.SizeOfImage)))
      {
/* On désassemble rapidement le code précédant l'adresse de retour */
      ReadProcessMemory(args->hProcess,(void *)(dwRetAddr-6),(void
*)&code,sizeof(unsigned char)*6,&nbr);
      /* FAR JMP */
      if ((code[0]==0xFF) && ((code[1]==0x15) || (code[1]==0x25)))
            bFound = 1;
      if ((code[4]==0xFF) && ((code[5] & 0xF0)==0xD0))
            bFound = 1;

      /* Fake CALL (PUSH/RET) */
      if (code[5]==0xC3)
            bFound = 1;

      /* Call simple */
      if (code[1]==0xE8)
            bFound = 1;
      }
      dwEsp+=4;
} while (bFound==0);
```

On tente ici de désassembler le code se trouvant avant l'adresse à vérifier, afin de déterminer si l'adresse récupérée dans la pile a bien été pushée par une instruction de contrôle (de type CALL). L'utilisation de ReadProcessMemory ne pose pas plus de problèmes que cela, car elle n'est pas intrusive, et donc considérée par la majorité des protections anti-malwares comme inoffensive.

Utilisation de l'injecteur

J'ai développé, afin d'illustrer cette technique, un injecteur sommaire [4], qui se contente d'injecter une Dll que j'ai conçue dans un processus cible afin de lui faire afficher une MessageBox indiquant "It Works !". Cet injecteur ne permet pas d'injecter du code arbitraire dans un processus cible, mais est juste là pour démontrer que la protection offerte par des protections anti-malwares telles que « A² Anti-Malware » est peu efficace face à ce type d'exploitation.

Conclusion

Cette nouvelle technique d'injection est plus furtive que les précédentes, et permet de contourner les protections anti-malware actuelles afin d'injecter du code dans des processus. Les tests ont été réalisés sur Windows XP Pro SP2, et Windows 2003 Server. Elle suppose cependant que la protection anti-malware autorise l'injecteur à débogueur le processus ciblé, condition sine-qua-none pour une injection réussie.

Remerciements

Je tiens à remercier Eloi « Baboon » V. pour son aide précieuse lors des recherches concernant cette technique d'injection, et principalement pour son outil « IINNNJJJJ », développé en parallèle de cet injecteur.

Références

[1] http://en.wikipedia.org/wiki/DLL_injection
[2] http://www.codeproject.com/KB/DLL/DLL_Injection_tutorial.aspx
[3] http://www.ring3circus.com/rce/armadillo-nanomites-and-vectored-exception-handling/
[4] http://sysdream.com/articles/New-Injection-Method-PoC.rar
[5] http://baboon.rce.free.fr/