Алексей Синцов Digital Security, dookie@inbox.ru

«Каждое из наших самых прочных убеждений может быть опрокинуто или, во всяком случае, изменено дальнейшими успехами знания»
Т.Хаксли

УБИТЬ DEP'A

ТЕОРИЯ И ПРАКТИКА ОБМАНА HARDWARE-DEP

Сегодня твоему вниманию будет представлены методы обхода защиты, именуемой DEP. Некоторое время ее создатели были убеждены в надежности технологии, но их убеждениям суждено было рухнуть как карточный домик. Сейчас мы посмотрим на основные методы взлома DEP и умело применим их на практике, а по ходу событий выясним, насколько осуществимы эти методы в жизни.

В ПРЕДЫДУЩИХ СЕРИЯХ

В прошлой статье мы научились обнаруживать и эксплуатировать уязвимости ActiveX через интерфейс браузера типа IE6/IE7, использовать известную уязвимость на переполнение буфера в компоненте QuickSoft EasyMail Object и даже нашли новую уязвимость небезопасного метода чтения, которая может привести к нарушению конфиденциальности и утечке чувствительной информации. Напомним, что вызов функции SubmitToExpress() со строкой более 256 байт переписывает адрес возврата, регистр ESI, а также указатель и дескриптор SEH.

```
cccc...260...ccccAAAAffffBBBBffffffff
ffffffffffffffffffffffffDDDD

ESI = AAAA
RET = BBBB
SEH = DDDD
```

Мы написали два эксплойта, выполняющие heap-sprray и передающие управление через SEH дескриптор и через вызов CALL [ESI+CC] в коде уязвимой программы. Продолжим изучение атак на клиентов через браузер на том же примере, но теперь усложним задачу, добавив защиту DEP (Data Execution Prevention), которую мы попробуем обойти, используя вышеуказанную уязвимость переполнения буфера в стеке. Ни безопасные методы ActiveX, ни DEP, ни другие технологии, вроде ASLR (Address space layout randomization), понятное дело, спасти от этих бед не могут. Тут поможет только грамотное распределение прав.

WHO IS MISTER DEP?

Многие из Вас уже конечно знают, что такое DEP и с чем его едят, но собрать всю информацию в один абзац будет полезно, так сказать для напоминания.

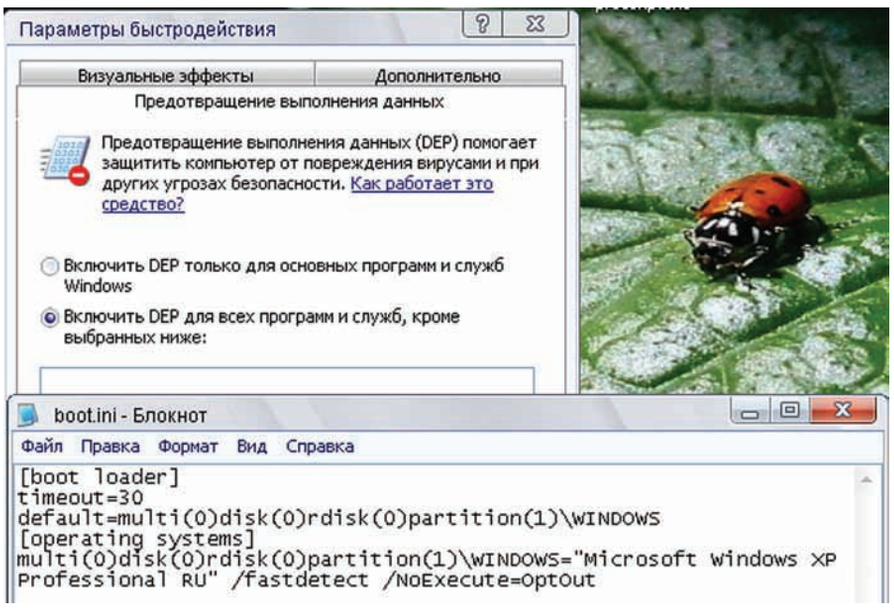
Итак, DEP — это технология, компании Microsoft, которая использует неиспользуемый в процессорах бит NX/XD (да, два названия... у AMD — NX, у Intel — XD) для «отметки» областей памяти. То есть, если в таблице разметки страниц памяти этот бит будет установлен, то код в данной области не является исполняемым. И если каким-то чудом, регистр EIP указывает в такую область, то генерируется исключительная ситуация и приложение с шумом вылетает (если, конечно, программист не установил обработчик). Соответственно, для того чтобы активировать DEP нужно иметь процессор с поддержкой NX/DX бита и ОС Windows с поддержкой этой технологии (>= Windows XP SP2). Также Microsoft позаботилась о тех обездоленных, у которых такого процессора нет — они смогут включить так называемый software-DEP. Только это не совсем так. Модное буквосочетание то же, но по факту — это другая технология, не связанная с битом и/или его эмуляцией. В данном случае просто используется защита от перезаписи SEH дескриптора. Изначально эта технология называлась SafeSEH, и ее просто переименовали и подогнали под решение DEP.

ACCESS VIOLATION

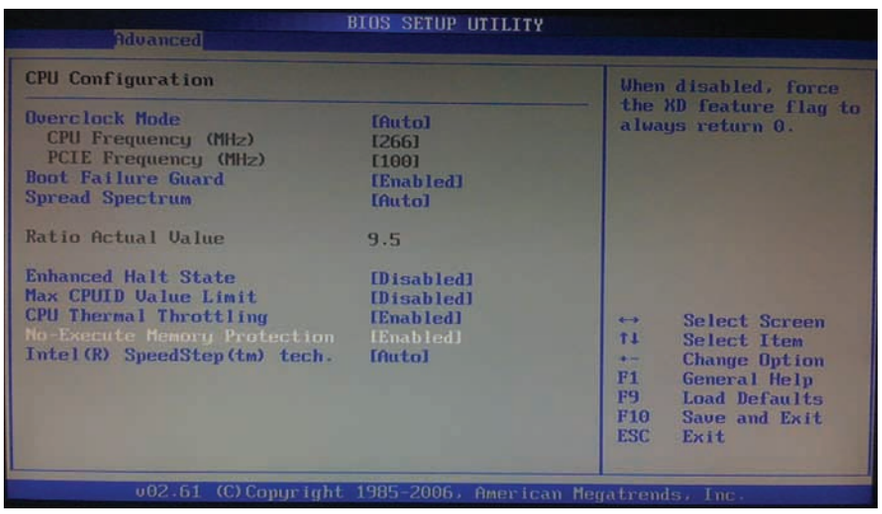
Что будет с нашими эксплойтами из предыдущей статьи, если включить DEP? Как сработает защита? Я отвечу на эти вопросы, но сначала расскажу, как запускать DEP. Для начала необходимо убедиться, что наш процессор поддерживает эту технологию. Зайдем в BIOS и посмотрим во вкладках, поддерживается ли необходимый бит защиты. Мой Intel Core2 Duo — поддерживает, но по умолчанию отключен. Это значит, что в таком варианте, можно использовать software-DEP — и очень легко обмануться (купил процессор с NX/XD битом,

врубил DEP, а это вовсе и не тот DEP :). Кстати, по умолчанию в Windows XP, DEP также отключен для всех процессов кроме самых важных. И IE6/IE7 таким процессом не считается. Это было сделано из соображений совместимости. Я не поленился и включил DEP для всех процессов. Сделать это можно в файле C:\boot.ini, или в свойствах «Моего Компьютера», вкладка «Дополнительно», кнопка «Параметры» в зоне «Быстродействие», а там вкладка «Предотвращение выполнения данных» — далее «Включить DEP для всех программ и служб». Данный интерфейс является Front-End'ом для редактирования соответствующего параметра в C:\boot.ini. Именно в этом файле задается политика DEP:

```
/noexecute=OptIn — значение по умолчанию для XP/Vista. DEP только для системных процессов
```



Настройки DEP



Включаем DX бит

```
/noexecute=OptOut — по умолчанию в Windows Server 2003 SP1. DEP для всех процессов, кроме указанных пользователем.
/noexecute=AlwaysOn — DEP для всех, нет исключений.
/noexecute=AlwaysOff — DEP отключен для всех (даже для системных компонент).
```

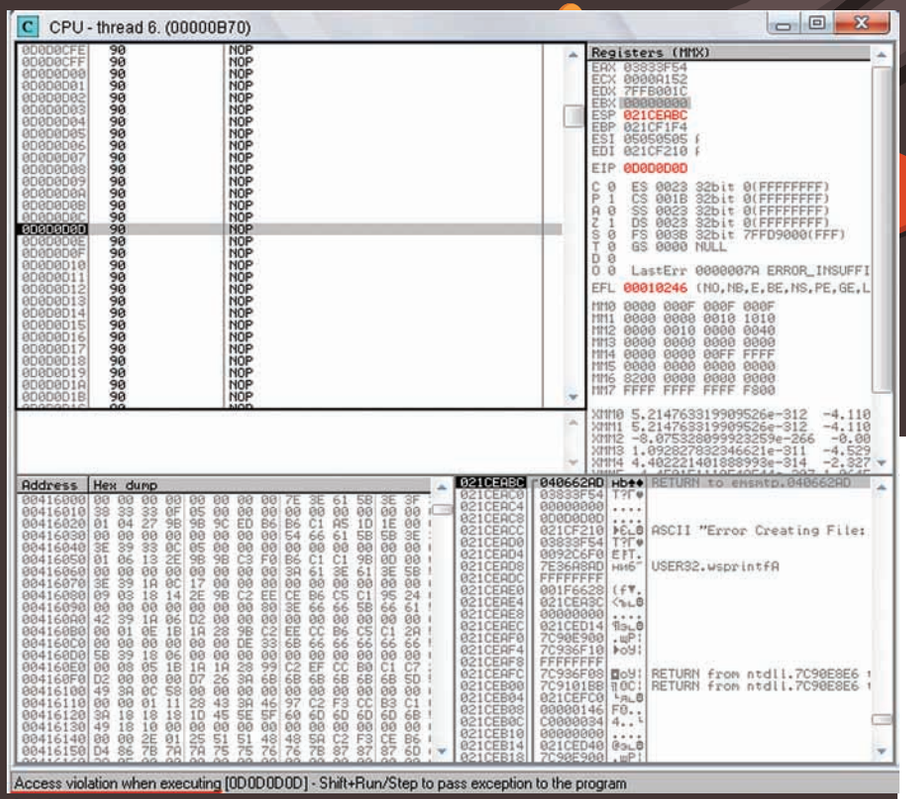
Потом перезагрузка. Посмотреть результат работы можно SysInternals Process Explorer'ом товарища Марка Руссиновича. В случае software-DEP и политики OptOut, мой IE7 становится под защиту. И правда, эксплойт из предыдущего номера, который использует SEH для выхода на шеллкод в куче, не сработал, так как защитный механизм определил, что адрес переписанного дескриптора не находится в его списке. Однако данная защита достаточно надежно обходится и есть много материалов на английском и русском языке. Я не буду тут их описывать, хотя бы потому, что в нашем случае мы еще контролируем регистр вызова CALL и адрес возврата. Таким образом, кроме перезаписи SEH у нас есть еще два варианта передачи управления на кучу с шеллкодом. Например, второй эксплойт (есть на диске прошлого номера), использующий вызов «CALL [ESI+CC]» в уязвимом коде, успешно работает в режиме software-DEP. Кстати, предлагаю читателю самостоятельно усовершенствовать второй эксплойт, сделав так, чтобы он делал не две разные кучи, а одну большую как в первом SEH-эксплойте (в прошлом номере я описывал упрощенную генерацию heap-spray.

В общем, случай с software-DEP, неинтересен для нас, поэтому рассмотрим hardware-DEP (используя процессор с поддержкой бита NX/XD). В таком варианте, что любопытно, SEH уже не проверяется на изменение, но и шеллкод также не исполняется. В обоих вариантах эксплойта, управление успешно перешло в кучу с шеллкодом, однако на первом же NOP'е возникла исключительная ситуация — Access violation when executing [0D0D0D0D]. А произошло это потому, что страница памяти под кучу, созданная с помощью JavaScript heap-spray, помечена как неисполняемая. В этом можно убедиться, открыв в дебаггере карту памяти процесса и посмотреть, что напротив наших кучек нет буковки «E» в столбце «Access».

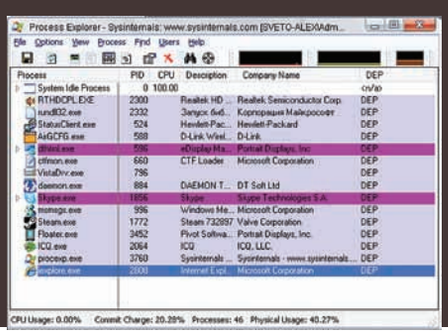
DEP IS DEAD

В Сети сложено много материала, на тему обхода DEP. В основе всех техник лежит концепция, получившая название «ret2libc». Раз мы можем передавать управление только в исполняемые области памяти, то воспользуемся этим. Ведь есть много полезных функций, код которых, естественно, лежит в исполняемой области, например, WinExec. Достаточно, заменить адрес возврата на адрес функции WinExec и передать через стек пару параметров — и все — мы выполнили запуск приложения! Конечно, это удобно для демонстрации уязвимости, но практическое применение достаточно ограниченно — одно дело выполнить несколько функций, и совсем другое связать их ввод-вывод, как например, инициализация/открытие сокета, обработка соединений и организация ввода/вывода для cmd.exe (классический шеллкод). Такую задачу, одними вызовами функций не решить.

Однако это уже что-то. Так, например, можно сделать несколько последовательных вызовов функций и что-то сделать с текущим процессом. В 2005 году был предложен простой метод обхода DEP и передачи управления шеллкоду. Вместо адреса возврата, подставлялся адрес функции VirtualAlloc() — для выделения памяти. Причем в параметрах для этой функции указывается, что память нужно пометить как исполняемую, кроме того мы не должны выделять новый кусок, так как в таком случае мы не знаем адрес этой памяти (черт его знает, где он выделит кучу), мы «обновляем» память на уже выделенном сегменте, который не будет использоваться до шеллкода. Далее идет адрес функции memcpy(), которая копирует в выделенную нами память шеллкод. Завершает все действо адрес возврата из memcpy(), который как раз указывает на пересозданную область памяти с шеллкодом, который уже помечен как исполняемый. В нашем же случае, память уже выделена, и шеллкод там уже есть, и адрес мы знаем. Достаточно вызвать



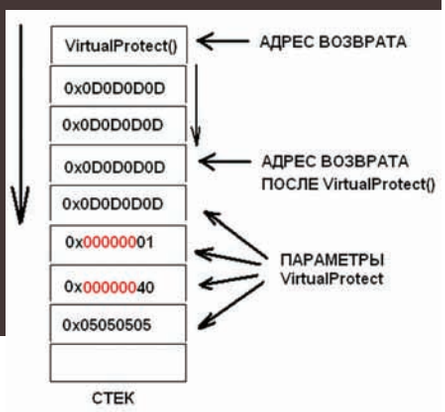
DEP не дает нам исполнить код из кучи



Process Explorer показывает процессы с DEP

функцию VirtualProtect(). Но, к сожалению, есть одно «но» — мы не можем передавать параметры с нулевыми байтами, а вызов VirtualProtect() предполагает передачу флага доступа (третий параметр, флаг должен быть 0x000040 — RWX) и размер памяти. Эти параметры должны содержать нули.

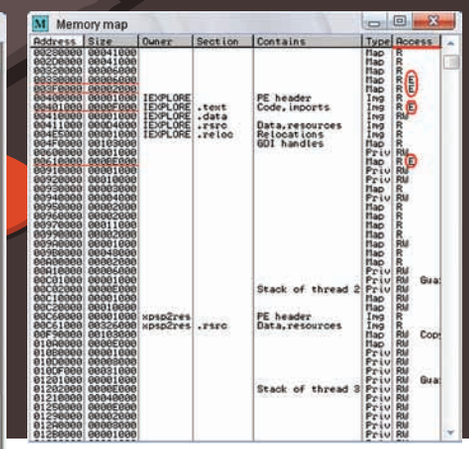
```
VirtualProtect(
  IN    LPVOID lpAddress,
// указатель на адрес памяти -
0x0D0D0D0D

  IN    SIZE_T dwSize,
// Размер памяти — 0x1

  IN    DWORD flNewProtect,
// флаг - 0x40

  IN    PDWORD lpflOldProtect
// указатель, на память, куда
запишется ответ(старые флаги),
пусть 0x05050505
);
```



Идея с VirtualProtect

Бороться с нулевым байтом дело неблагодарное. Что еще нам предоставляет Windows API для борьбы с DEP'ом? В Windows XP SP3 (у меня как раз такая) появилась новая API функция — SetProcessDEPolicy(), которая может просто-напросто отключить DEP. Да, но опять-таки, в качестве параметра должен быть ноль. Опять мимо. На самом деле SetProcessDEPolicy() является всего лишь оберткой для функции NtSetInformationProcess():

```
NtSetInformationProcess
  (
  IN HANDLE ProcessHandle,
  // Дескриптор процесса, 0xff —
текущий процесс

  IN PROCESS_INFORMATION_CLASS
ProcessInformationClass,
 // 0x22 — смена флагов процесса
```

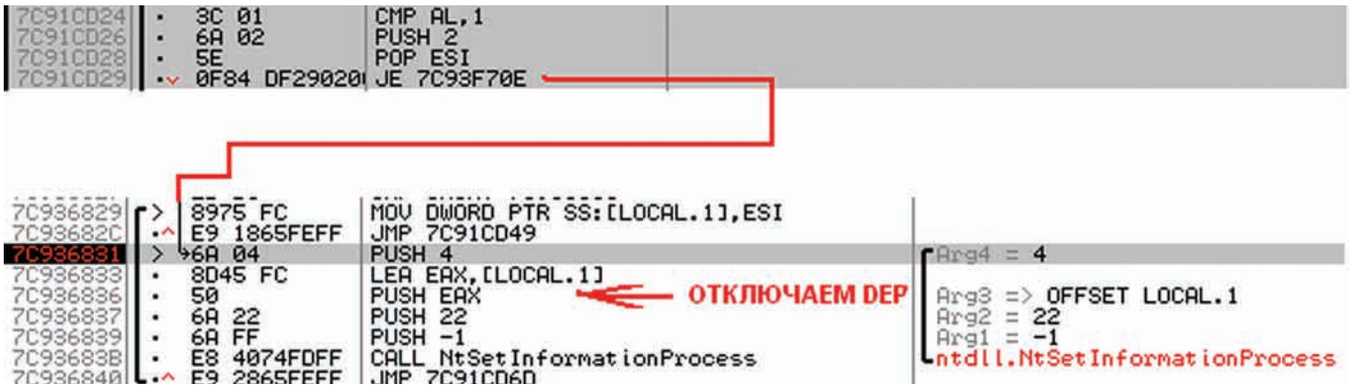


Исполняемых участков не так уж много

```
  IN PVOID ProcessInformation,
// Указатель добавляемый флаг
чтобы отключить DEP для про-
цесса, указатель должен быть на
0x0000002

  IN ULONG
ProcessInformationLength
// размер флага — (0x4) 4 байта
);
```

Но и эту функцию мы не можем вызвать по тем же причинам — зловещие нулевые байты! Казалось бы, что еще можно найти? И вот бравые парни «Skape» и «Skywing» нашли такое место в коде ntdll.dll:

```
Address1:
cmp al,0x1 ; младший разряд EAX=1 ?

push 0x2 ; заносим в стек 0x2
(Самое главное)
pop esi ; вынимаем из стека, что
там есть (0x2) и кладем в ESI
je LdrpCheckNXCompatibility + 0x1a
;Если младший разряд EAX=1 прыгаем
. . .
mov [ebp-0x4],esi  ; копируем
0x2(лежит в ESI) по адресу EBP-4
jmp LdrpCheckNXCompatibility +
0x1d  ;прыгаем дальше

. . .

; проверяем, равен ли наш пара-
метр 0? (мы только что скопирова-
ли туда 0x2)
cmp dword ptr [ebp-0x4],0x0

jne LdrpCheckNXCompatibility+0x4d
; ну раз 4!=2, прыгаем дальше

. . .

push 0x4 ; в стек 0x4

lea eax,[ebp-0x4]  ; в EAX адрес
ebp-0x4, то есть указатель на 0x2
```



DEP можно просто-напросто отключить

```
push eax ; кладем указатель на
0x2 в стек
push 0x22 ; 0x22 в стек
push 0xff ; 0xff (-1) в стек
call NtSetInformationProcess
; вызов нужной функции, с нужными
; параметрами — отключение DEP
jmp   LdrpCheckNXCompatibility +
0x5c  ; прыжок ...

. . .

pop esi
leave  ; удаляем локальный стек
ret     0x4 ; берем адрес возвра-
та со сдвигом в 4 байта
```

Если в качестве адреса возврата указать адрес этого куска кода, то процессор, выполнив первый прыжок (а для этого в AL должна быть единица) отключит DEP и доберется до следующего адреса возврата из стека, который будет указывать на кучу с кодом. Тут надо только заметить, что адрес возврата должен быть с некоторым сдвигом, так как «LEAVE» удалит большой сегмент стека, а, по сути, сделает ESP = EBP. Главное, чтобы младший регистр EAX был равен 1. Опять же, для этого можно первоначально прыгнуть на код, который устанавливает этот регистр в значение 1. Другими словами, переписываем адрес возврата, адресом функции, которая заносит в AL единицу, например в той же ntdll.dll:

```
. . .
Address2
mov     al,0x1
ret     0x4
```

Тогда входной буфер будет, типа:

```
cccc…260…ccccAAAAffffBBBBCCCCXXXXX
XXX…100…XXXXXXXXXXX
AAAA=0x05050505
BBBB=Address2
CCCC=Address1
X=0x0D
```

FIGHT!

С помощью OllyDbg выполняем аттач (File->Attach) к процессу iexplore и пытаемся найти в карте памяти секцию .code в ntdll.dll (View->Memory). Открываем ее в дизассемблере и ищем последовательность команд (Ctrl+S):

```
al,1
retn 0x4
```

Это будет адрес Address2. Аналогично ищем Address1 по последовательности команд:

```
cmp     al,0x1
push    0x2
pop     esi
```

Теперь надо довести программу до адреса возврата без проблем, то есть, чтобы не было исключительных ситуаций. Анализируя ассемблерный код видим, что зависимость прыжков зависит от регистра ESI. В первый раз, там, где возникает исключительная ситуация, при «CMP [ESI+180],1». Потом идет такой код:

```
xor ebx, ebx ; обнуление
push -1
cmp [ESI+CC],EBX ; сравнение с 0
```

В зависимости от этого сравнения идет либо вызов «CALL [ESI+CC]», либо нет. Нам лишних вызовов не надо, поэтому нам надо, чтобы по адресу ESI+CC был 0. Таким образом, если там будет 0, программа постепенно дойдет до ret и выйдет в вызывавшую ее функцию:

```
call emsmtp.026c6232 ; только что
отсюда благополучно вышли…
xor eax,eax ; обнуляется результат

pop edi ; восстанавливаются реги-
стры

pop esi
pop ebx
leave          ; чистится стек
retn 0x8       ; указатель на НАШ
адрес возврата (AAAA)
```

Теперь проблем нет — мы делаем две кучи, как в эксплойте из прошлого номера, в первой куче одни нули, во второй — NOP'ы и шеллкод. Регистр ESI переписываем 0x05050505, указатель на 0 из первой кучи, а адрес возврата — по указанной схеме, только увеличивая разницу между BBBB и CCCC, так при выходе на BBBB у нас retn 8. Итоговый буфер:

```
cccc…260…ccccAAAAffffBBBBffffffffC
CCCXXXXXXXXX…100…XXXXXXXXXXX
```

Но здесь возникают две проблемы. Во время чистки стека, перед выходом на первый адрес возврата EBP становится 0x4646464646 — мусором, который указан в буфере перед адресом возврата(BBBB). В таком варианте мы не сможем вызвать функцию отключения DEP, так как там EBP используется для хранения 0x2:

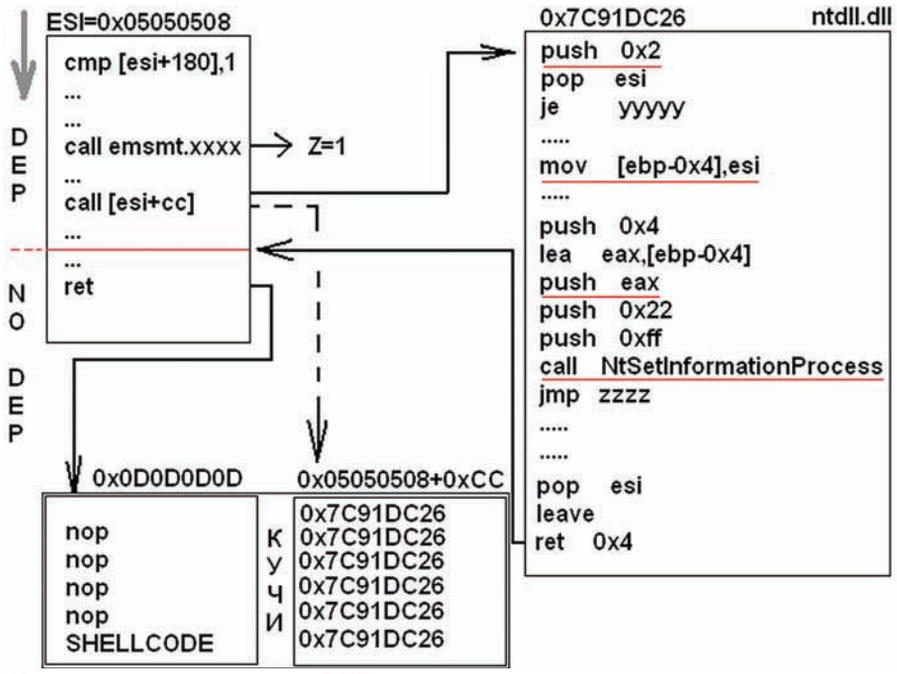
```
mov     [ebp-0x4],esi
```

Это можно обойти, указав адрес из кучи, но тогда, потом куча станет стеком, когда выполнится «leave» после NtSetInformationProcess! Но есть еще одно «но» — адреса возврата, которые мы подсунули в стек (BBBB и CCCC), не соответствуют тем, что мы запихнули в буфер. Дело в том, что наш ActiveX понимает только ASCII набор байтов. Все что больше 0x7C, ActiveX превращает в значки «?» — 0x3F. Эта проблема сильно подрывает задуманную акцию, так как нужные нам адреса имеют значащие разряды со значением больше чем 0x7C. Я почти было отказался от идеи демонстрирования обхода DEP'а, на примере этого ActiveX. Однако вспомним, что у нас есть один «CALL [ESI+CC]» в коде программы, который мы можем использовать. Для попадания в этот вызов, достаточно, чтобы по адресу ESI+CC не было нулевого значения, что само собой решается, ведь мы заносим туда адрес. Этот адрес берется из кучи, в которой могут быть любые байты! Но CALL у нас всего один. Мы можем вызвать только одну функцию. Логично предположить такой вариант: с помощью CALL мы отключаем DEP, а потом адресом возврата прыгаем на кучу с шеллкодом, и уже без проблем выполняем код. Но перед вызовом CALL у нас AL не равен единице, что означает, что при «je LdrpCheckNXCompatibility+0x1a» мы не перейдем в прыжок и не отключим DEP. Но нет худа без добра, ведь перед тем как делать CALL, у нас идет вызов функции из ActiveX, которая перед своим завершением вызывает функцию MultiByteToWideChar(). В ней последнее сравнение заканчивается так, что бит Z становится равным единице. Что это значит? А это значит, что нам не нужно вообще сравнение, чтобы перейти по необходимому je. У нас уже Z=1, и

неважно, где было сравнение. Таким образом, мы настроим прыжок не на «CMP AL,1», а на 2 байта выше, туда, где уже «PUSH 2». Таким образом, далее мы перейдем по je и выполним отключение DEP. После этого, код успешно доберется до адреса возврата и перейдет на кучу с шеллкодом. Итоговый буфер:

```
cccc...260...ccccAAAAffffBBBB
```

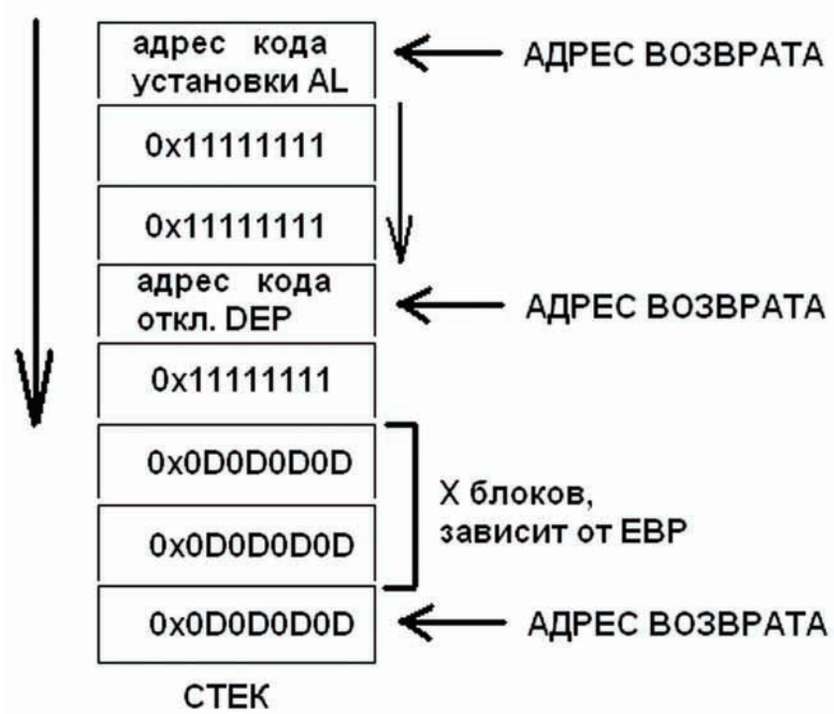
AAAA = 0x05050505 — указатель на первую кучу, которой лежит Address1
BBBB = 0x0D0D0D0D — адрес возврата, на вторую группу куч с шеллкодом
Теперь, собственно, нужно подставить Address1 в первую группу куч. В моем случае, адрес — 0x7C91CD26. Но он зависит от версии ntdll.dll. Если этот адрес указать неверно, то эксплойт с большой вероятностью не обвалится, а попробует выполнить шеллкод, но уже без отключенного DEP'а. Кроме того, нужно учитывать все сдвиги в самой куче, относительно ее начала. Если с адресом 0x0D0D0D0D нет косяков, так как все разряды одинаковые, то в данном случае можно столкнуться с той проблемой, что по адресу, на который указывает CALL[ESI+CC] (0x050505D1) будет лежать, например, 0x267C91CD. Чтобы не ошибиться с адресом кучи, я предлагаю произвести расчет: заголовок кучи — 36 байт. Наш «спам» адресов начинается через 36 байт, после начала базового адреса. Спамим мы в аккурат по 4 байта. То есть, рассчитать адрес надо так, чтобы попадание было только при наличии базового адреса (он не бывает одинаковым каждый раз, тем более на разных системах). Однако, по той же карте памяти можно заметить, что при выделении больших



Итоговая схема атаки на DEP

блоков памяти, больше двух младших разрядов, каждая новая куча будет начинаться с 0xXXYY0000. И логично предположить, что наш адрес начинается с 0xXXYY0024 (и так далее + 4 байта). Поэтому последний разряд должен быть кратен 0x4, а так как у нас +0xCC и получается, что разряд становится 0xD1. Поэтому увеличиваем исходный адрес на 0x3 и тогда внедрять мы должны 0x05050508. При написании эксплойта не забываем, что у нас little-endian порядок байтов.
В итоге рождается хороший эксплойт (его в целях ознакомления ты можешь лицезреть на нашем DVD).

Идея с NtSetInformationProcess



ВЫВОДЫ

В данной статье мы рассмотрели некоторые виды атак на DEP и даже попробовали одну из них в реальных условиях, изменив ее под реалии эксплуатации уязвимости. Также очевидно, что исследование кода приложения помогает понять как можно использовать те или иные возможности уязвимого кода, усиливая эффект от атаки, а в нашем случае мы даже упростили отключение DEP. Так, например, тот факт, что Z бит всегда установлен в единицу перед вызовом CALL с контролируемым через кучу адресом, позволил нам модифицировать алгоритм атаки так, что назвать ее классической уже нельзя, хотя применяется все тот же ret2libc подход. Однако показанные механизмы не будут работать в ОС со случайной адресацией памяти (ASLR), так как мы не будем знать адреса функций VirtualProtect или NtSetInformationProcess. При каждом запуске процесса — эти адреса будут меняться. Кроме того, в IE8 DEP перманентный. Это значит, что при запуске IE8 ставит себе DEP по умолчанию (с помощью функции SetProcessDEPPolicy). Это уже исключает возможность снятия DEP-флага с процесса методами ret2libc, так как в таком случае NtSetInformationProcess вернет ошибку с отказом в доступе. Но и это уже не проблема, ведь буквально в начале февраля, когда эта статья только задумывалась, на BlackHat 2010 DC, Дионисисом Блазакисом (Dionysus Blazakis) была продемонстрирована атака на IE8 с ASLR(случайная адресация) и DEP. Атака использует возможности компиляторов ActionScript или Java, которые помещают скомпилированный код в исполняемые участки памяти. Этот метод получил название JIT-spray, но это уже совсем другая история...