

Способы обнаружения атомов¹.

Ранее описан способ блокировки исполнения атома - изоляция выборки данных(**DF**) через блокировку памяти. При обнаружении **DF** адресация изменяется, но так как в атоме **DF** не существует, он не может получить доступ к данным. Далее рассмотрены методы не блокировки, а обнаружения атомов.

Для обнаружения атома необходимо в начале получить событие **DF**. Далее можно выполнить валидацию **DFG** или иные манипуляции. Базовые способы получения **DF**:

1. Раскодировка адресов для каждой инструкции, это получение линейного адреса через раскодировку **MODRM** инструкции. Для этого необходимо обработать поток инструкций — выполнить трассировку или эмуляцию.
2. Установка ловушки. Срабатывание ловушки является событием **DF**. Ловушка ставится хардверно, через блокировку памяти или установку аппаратных точек останова(не универсально). В первом случае необходимо перенаправить **DF** на оригинальный адрес(**IDP**), что бы использовать ловушку повторно.

Помимо базовых способов можно найти системное решение. Так например обнаружить **DF** можно по расширению стека(доступ к сторожевым страницам). Кроме этого система реализует два монитора памяти: **WW**(*write watch*) и **WSW**(*working set watch*). Первый(**WW**) работает со специальной памятью(**MEM_WRITE_WATCH**). Второй(**WSW**) механизм универсален. Это монитор рабочего набора. Лог добавленных страниц доступен из **UM: ProcessWorkingSetWatch**.

При первом обращении к странице памяти она добавляется в рабочий набор, это событие записывается в лог. Так как далее страничных нарушений не будет при обращении к данной странице, то **DF** не будет логгироваться. Для повторного мониторинга страницы весь рабочий набор должен быть сброшен: **EmptyWorkingSet()**. Но после такого сброса последует активная подгрузка страниц и соответственно сохранение этих событий в лог.

WSW это второй(с **ProcessHandleTrace**) механизм **NT**, который реализует раскрытие ядерных адресов. Пример лога для **NtQueryInformationProcess** с буфером(0x360000), не присутствующем в рабочем наборе:

0040109F		6A 0F	push 0xF
004010A1		6A FF	push -0x1
004010A3		E8 48000000	call <jmp.&ntdll.ZwQueryInformationProcess>
004010A8		6A FF	push -0x1
0013EFC0	80615A83		
0013EFC0	00360001		
0013EFC4	80615A85		
0013EFC8	004010A3	T.004010A3	
0013EFC8	80615A85		
0013EFD0	004010A9	T.004010A9	
0013EFD4	80615A85		
0013EFD8	004010A3	T.004010A3	
0013EFD8	00000000		
0013EFD8	00000000		

WSW лог это массив записей:

```
typedef struct _PROCESS_WS_WATCH_INFORMATION {
    PVOID FaultingPc;
    PVOID FaultingVa;
} PROCESS_WS_WATCH_INFORMATION, *PPROCESS_WS_WATCH_INFORMATION;
```

¹ DFG.pdf: Защита потока данных(Data Flow Guard).

Как видно первая запись в логе это событие **DF** в ядре при доступе к аргументу сервиса.

Следующие записи являются результатом работы отладчика(0x80615A85 -> 0x53EA85(смещение при загрузке ядра в *OllyDbg*, для загрузки необходимо изменить тип подсистемы Native -> Gui):

```
KernelMode - ntkrnlpa.exe - [*G.P.U* - main thread, module ntkrnlpa]
File View Debug Plugins Options Window Help
[Icons] [L] [E] [M] [T] [W] [H] [C] [K] [B] [R] [S] [Icons] [?]

0053EA4A ProbeForWrite | $ 8BFF | mov edi,edi
0053EA4C | . 55 | push ebp
0053EA4D | . 8BEC | mov ebp,esp
0053EA4F | . 8B45 0C | mov eax,dword ptr ss:[ebp+0xC]
0053EA52 | . 85C0 | test eax,eax
0053EA54 | .v 74 49 | je short ntkrnlpa.0053EA9F
0053EA56 | . 8B4D 10 | mov ecx,dword ptr ss:[ebp+0x10]
0053EA59 | . 56 | push esi
0053EA5A | . 8B75 08 | mov esi,dword ptr ss:[ebp+0x8]
0053EA5D | . 49 | dec ecx
0053EA5E | . 85CE | test esi,ecx
0053EA60 | .v 75 37 | jnz short ntkrnlpa.0053EA99
0053EA62 | . 8D4406 FF | lea eax,dword ptr ds:[esi+eax-0x1]
0053EA66 | . 3BF0 | cmp esi,eax
0053EA68 | .v 77 28 | ja short ntkrnlpa.0053EA92
0053EA6A | . 3B05 34B14800 | cmp eax,dword ptr ds:[MmUserProbeAddress]
0053EA70 | .v 73 20 | jnb short ntkrnlpa.0053EA92
0053EA72 | . BA 00F0FFFF | mov edx,-0x1000
0053EA77 | . 23C2 | and eax,edx
0053EA79 | . B9 00100000 | mov ecx,0x1000
0053EA7E | . 03C1 | add eax,ecx
0053EA80 | . 57 | push edi
0053EA81 | . 8BF8 | mov edi,eax
0053EA83 | > 8A06 | mov al,byte ptr ds:[esi]
0053EA85 | . 8806 | mov byte ptr ds:[esi],al
0053EA87 | . 23F2 | and esi,edx
```

Тот же вызов сервиса но при сброшенном рабочем наборе:

```
004010DF | . 6A 0F | push 0xF
004010E1 | . 6A FF | push -0x1
004010E3 | . E8 00000000 | call <jmp.&ntdll.ZwQueryInformationProcess>
004010E8 | . C9 | leave

0013EFC0 00615A85 T.004010A9
0013EFC0 004010A9 ntdll.KiFastSystemCallRet
0013EFC4 7C90E4F4 ntdll.7C90E4F5
0013EFC8 7C90E4F5 ntdll.KiFastSystemCallRet
0013EFC0 0013EFC4 ntdll.KiFastSystemCallRet
0013EFD0 0013EFD4 7C90DC8C RETURN to ntdll.ZwSetInformationProcess+0C
0013EFD0 7C90DC8D ntdll.7C90DC8D
0013EFD0 76BE227B RETURN to psapi.76BE227B from ntdll.ZwSetInformationProcess
0013EFE0 76BE227B RETURN to psapi.76BE227B from ntdll.ZwSetInformationProcess
0013EFE4 004010AC RETURN to T.<ModuleEntryPoint>+92
0013EFE8 004010AD T.004010AD
0013EFC0 004010F6 Entry address
0013EFF0 0040200D T.0040200D
0013EFF4 7C90DC8A ntdll.7C90DC8A
0013EFF8 7FFE0301
0013EFC0 00615A83

0013F000 00360001
0013F004 805B40D2
0013F008 7FFDF035
0013F00C 805B40D2
0013F010 7FFDE035
0013F014 805B40D2
0013F018 0040003D T.0040003D
0013F01C 805B40D2
0013F020 1000003D EMET.1000003D
0013F024 805B40D2
0013F028 5D07003D sh.ineng.5D07003D
0013F02C 805B40D2
0013F030 76BE003D psapi.76BE003D
0013F034 805B40D2
0013F038 77C0003D msuvcrt.77C0003D
0013F03C 805B40D2
0013F040 77DC003D advapi32.77DC003D
0013F044 805B40D2
0013F048 77E7003D report4.77E7003D
0013F04C 805B40D2
```

Видна последовательность выполнения кода — вызов сервисного шлюза, возврат на него, **DF** из буфера и активность **EMET**. Следует заметить что **DF** из буфера происходит в **ProbeForWrite()** - это валидация аргументов сервиса. Под **WOW** в логе сохраняются **UM** адреса слоя виртуализации.

Этот(**WSW**) логгер хорошо подходит для обнаружения атомов. Он пригоден для следующих приложений:

1. Непосредственно обнаружение атомов.
2. Обнаружение отладочной активности.
3. Обнаружение **КМ** фильтров.

Упомянутый выше механизм трассировки описателей(*Handle Trace*) использовался для обнаружения ядерной фильтрации, что было реализовано в моторе **SHDE**(2012):

```
; о Валидация ядерной SFC для определения наличия фильтров.
;
; Первый адрес на стеке ядра это kss60, инструкция следующая за
; процедурным ветвлением kssdoit, вызывающим NT-вектор. Второй адрес
; принадлежит телу NTAPI. Модель вызова
; сервисов (KiSystemService()/KiFastCallEntry()) не изменяется. Если
; вектор в SST направлен в фильтр, либо начало сервиса
; пропатчено (сплайсинг), то второй адрес будет принадлежать не
; NTAPI, а фильтру. Возможно что фильтров несколько, в этом случае
; SFC удлинится. Возможно фильтр не стоит на NTAPI, а находится
; глубже, например в менеджере объектов. В этом случае он не следит
; за вызов сервисов непосредственно (eg.: DrWeb ставит фильтр на
; OBTYPE.OpenProcedure). Однозначно определить NTAPI-фильтр можно
; только первый. Косвенная фильтрация не может служить для
; однозначной детекции.
```

Аналогично может быть использован и механизм **WSW**. Серию **DF** из области памяти нельзя отследить через **WSW**, но можно использовать цикл перезапусков целевого кода со сбросом рабочего набора, что даст все адреса, код по которым выполняет **DF**.

WSW позволяет исключить сервисы из валидации, так как отслеживает **DF** в них(сервисы являются атомами).

Перейдём к рассмотрению непосредственно детекта атомов. При обнаружении атомов трек **DF** является опциональным дополнением к **DFG** валидации через **WSW**:

1. Если нет записи **WSW** и данные изменились, то исполнялся атом.
2. Если есть запись **WSW**, то **DF(WSW)** должна быть из целевого кода или из сервиса. Такую проверку можно выполнить при трассировке целевого кода(таким образом выделив все адреса, по которым были **DF**). Может быть выполнена дополнительная проверка **DF**(раскодировка адресов) на соответствие **WSW**.

Например в **AVVM** сложные **API** являются заглушками для вызова атомов.

Если атом вызывается без обращений к аргументам до вызова его, то **WSW** будет пуста(1).

Если до вызова атома имеются **DF**, то он будет обнаружен через трассировку(2).

Если **WSW** эмулируется, то валидацию можно выполнить через трассировку(2).