

25-ЛЕТНИЙ ПАТЕНТ INTEL, КОТОРЫЙ УБИЛ ВАШИ ДИЗАССЕМБЛЕРЫ.

Всем привет, давно не было материала, и сегодня мы это исправим.

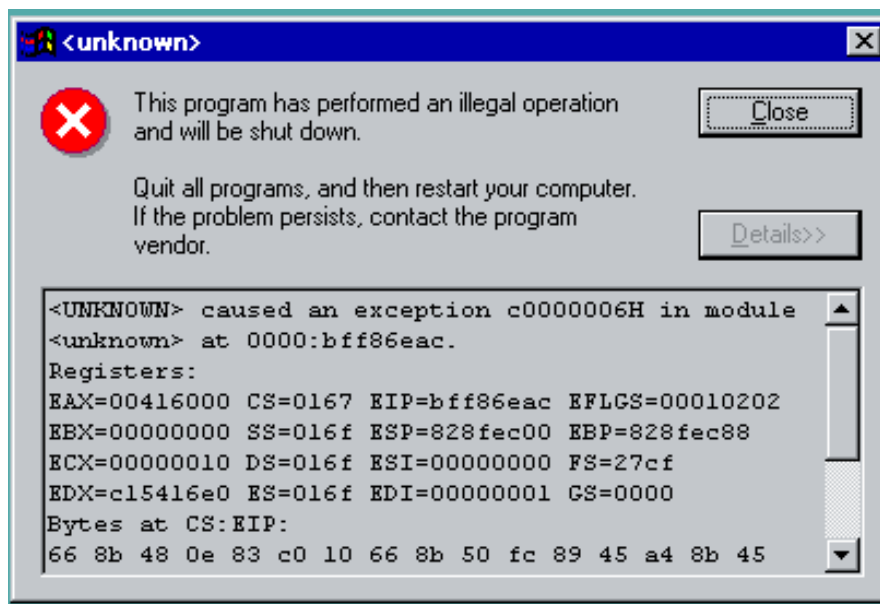
Наша история началась с сообщения от друга-реверсера colby57: *"Первый раз когда я это увидел, думал что сейчас уже улечу с крашем, но всё валидно"*. В приложении был образец шелла. Ничего особенного на первый взгляд, но посреди кода дизассемблер просто обрывался, показывая дальше "мусорные" байты. Процессор же этот код исполнял без проблем.

Этим "мусором" оказались два опкода: **OF 1A** и **OF 1B**. На самом деле их больше, но будут упомянуты всего два.

И это не баг в конкретном семпле. Это системный провал всей индустрии реверс-инжиниринга, заложенный патентом Intel почти 30 лет назад.

Копаем в прошлое: откуда взялись эти призраки?

Корни нашей проблемы уходят в далекий 1997 год, в один из самых элегантных патентов Intel, созданный в то время, когда мир готовился к проблеме Y2K. Тогда Intel столкнулась с вечной головной болью — обратной совместимостью. Инженеры добавляли в новые процессоры пачки крутых инструкций, но разработчики их не трогали, боясь, что их программа не запустится на компьютере, вышедшем пару лет назад.



Решение, предложенное в патенте **US5,701,442**, было гениально простым.

"А давайте мы зарезервируем целый диапазон опкодов — от **0x0F18** до **0x0F1F** — как просто заглушки?". В патенте их называли **Hintable NOPs**.

На процессорах тех времен они не делали абсолютно ничего. Но в любой момент любая заглушка могла "ожить" и превратиться в новую, полезную инструкцию. На старых компьютерах такая инструкция исполнялась бы как безобидный NOP, на новых — как мощный инструмент.

Так и случилось. **0x0F18** превратился в семейство инструкций **PREFETCH**, а **0x0F1E** спустя двадцать лет стал сердцем технологии защиты CET в виде **ENDBR64**. Большинство опкодов из этого диапазона со временем были либо задействованы, либо, по крайней мере, корректно распознаны дизассемблерами.

Кроме двух. Опкоды **0F 1A** и **0F 1B** стали настоящими призраками. Они остались в тени, забытые всеми, кроме самого кремния и страниц того самого патента.

Live-тест: ломаем всё

Хватит теории. По прежнему мой хороший друг написал простой PoC-бинарник, содержащий эти инструкции, и мы решили прогнать его по нашему паку стандартных инструментов. Давайте посмотрим на это вместе.

GitHub репозиторий с PoC

Первый пациент: x64dbg

Загружаем наш бинарник в отладчик. Вот они, наши "мусорные" байты. x64dbg честно показывает нам `???`, не в силах распознать инструкцию.

```
52      push edx
0F      ???
1B2468  sbb esp,dword ptr ds:[eax+ebp*2]
5A      pop edx
50      push eax
0F      ???
1A2489  sbb ah,byte ptr ds:[ecx+ecx*4]
58      pop eax
F7DA    neg edx
52      push edx
0F      ???
1B247B  sbb esp,dword ptr ds:[ebx+edi*2]
5A      pop edx
0F      ???
1A244B  sbb ah,byte ptr ds:[ebx+ecx*2]
F7D2    not edx
0F      ???
1A24E1  sbb ah,byte ptr ds:[ecx]
50      push eax
0F      ???
1A2476  sbb ah,byte ptr ds:[esi+esi*2]
58      pop eax
F7D2    not edx
50      push eax
0F      ???
1A2444  sbb ah,byte ptr ss:[esp+eax*2]
58      pop eax
52      push edx
0F      ???
1A2467  sbb ah,byte ptr ds:[ecx]
```

Стрелка EIP указывает на строку с байтами 0F 1A C0. В колонке дизассемблера на этой строке три вопросительных знака.

Но самое интересное — поведение процессора. Делаем шаг (F7), и... EIP спокойно перепрыгивает эту "неизвестную" инструкцию и идет дальше. Никакого краша, никакой ошибки. Для CPU это валидный, исполняемый код. А для отладчика — какой-то бред

Второй пациент: IDA Pro

А что же скажет любимая IDA? Открываем бинарник и видим удручающую картину.

```

.c57:0000000140007414 ; START OF FUNCTION CHUNK FOR sub_140001690
.c57:0000000140007414
.c57:0000000140007414 Loc_140007414: ; CODE XREF: sub_140001690+14↑j
.c57:0000000140007414 mov     rbx, 0E64B77E88C80000Ah
.c57:000000014000741E rol     rbx, 18h
.c57:000000014000741E ; END OF FUNCTION CHUNK FOR sub_140001690
.c57:000000014000741E
.c57:0000000140007422 dw 1B0Fh, 4C24h, 0C148h
.c57:0000000140007428 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_3. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007429 db 1Fh, 0Fh, 1Bh, 24h, 4Ch, 48h, 0C1h
.c57:0000000140007430 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_4. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007431 db 3, 0Fh, 1Bh, 24h, 4Ch, 48h, 0C1h
.c57:0000000140007432 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_5. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007433 db 8, 0Fh, 1Bh, 24h, 4Ch, 48h, 0C1h
.c57:0000000140007440 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_6. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007441 db 2 dup(0Fh), 1Bh, 24h, 4Ch, 48h, 0C1h
.c57:0000000140007448 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_7. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007449 db 15h, 0Fh, 1Bh, 24h, 4Ch, 48h, 0C1h
.c57:0000000140007450 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_8. PRESS CTRL-NUMPAD+ TO EXPAND]
.c57:0000000140007451 db 8, 0E9h, 57h, 0A2h, 2 dup(0FFh), 48h
.c57:0000000140007458 dq 7FFFFFFFFF8B9h, 241B0F08C9C14800h, 241B0F18C9C1484Ch
.c57:0000000140007470 dq 241B0F15C9C1484Ch, 4C241B0FC9D1484Ch, 4C241B0F14C1C148h
.c57:0000000140007488 dq 4C241B0F07C1C148h, 4C241B0F0AC1C148h, 4C241B0F0DC1C148h
.c57:00000001400074A0 dq 4C241B0F03C1C148h, 4C241B0F1EC9C148h, 4C241B0F12C9C148h
.c57:00000001400074B8 dq 0FA24EE90FC9C148h, 88CC0000AEB948FFh, 0F0CC1C14864B77Eh
.c57:00000001400074D0 dq 0F13C9C1484C241Bh, 0F0FC9C1484C241Bh, 0F16C9C1484C241Bh
.c57:00000001400074E8 dq 0E912C1C1484C241Bh, 800000B8FFFA228h, 4C241B0F06C8C180h
.c57:0000000140007500 dq 0C0C14C241B0FC0D1h, 19C0C14C241B0F05h, 0F18C0C14C241B0Fh
.c57:0000000140007518 dq 1B0F1EC0C14C241Bh, 0A21FE909C0C14C24h, 0C10000040B8FFFFh
.c57:0000000140007530 dq 0C8C14C241B0F1CC0h, 0CC8C14C241B0F80h, 0F16C8C14C241B0Fh

```

IDA не просто не понимает инструкцию. Она решает, что здесь **закончился исполняемый код** и начались данные. Анализ функции обрывается. Весь последующий код для статического анализатора просто перестает существовать. Это очень плохо, но это для неё обычное дело. @

Третий пациент: Binary Ninja & Ghidra

Здесь чуда тоже не произошло. Binary Ninja помечает инструкцию как **??**. Ghidra показывает **undefined**. Та же история — ведущие инструменты индустрии видят валидный код как ошибку

Address	Disassembly	Comment
1400070b2	int64_t sub_1400070b2()	
1400070b2	mov ecx, 0x400	
1400070b7	ror ecx, 0xc {0x40000001}	
1400070ba	??	
1400070bb		1b 24 4c c1 c1 .\$.L..
1400070c0	07 0f 1b 24 4c c1 c9 13-0f 1b 24 4c c1 c1 0f 0f	...\$.L.....\$.L....
1400070d0	1b 24 4c c1 c1 1e 0f 1b-24 4c c1 c9 10 0f 1b 24	\$.L.....\$.L.....\$
1400070e0	4c c1 c9 16 0f 1b 24 4c-c1 c1 07 e9 68 a0 ff ff	L.....\$.L.....h...
1400070f0	b8 00 00 00 ff c1 c8 07-0f 1b 24 4c c1 c0 05 0f\$.L....
140007100	1b 24 4c c1 c0 13 0f 1b-24 4c c1 c0 10 0f 1b 24	\$.L.....\$.L.....\$
140007110	4c c1 c8 08 0f 1b 24 4c-c1 c8 07 0f 1b 24 4c c1	L.....\$.L.....\$.L.
140007120	c0 0f 0f	...

Как же нам это использовать?

Так что же эти два опкода дают злоумышленнику на практике?

Самое очевидное — **сломать анализ**. Как мы видели в IDA, один такой NOP — и анализ функции обрывается. Весь последующий код для дизассемблера просто перестает существовать.

Пример уровня выше: **создание фейковых графов выполнения**. После `jmp` ставится комбинация `OF 1B E8...`, которую дизассемблер ошибочно примет за `CALL`, отправляя аналитика по ложному следу, который ведет в никуда.

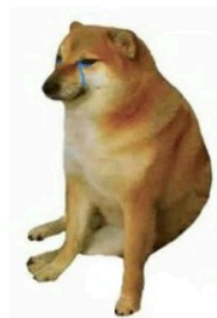
И это лишь верхушка айсберга, можно придумать тысячи разных применений.

Проблема индустрии. Проблема человека.

Но все эти трюки — лишь следствие. Настоящая проблема в том, что фундаментальная, документированная часть архитектуры x86 оставалась невидимой для наших самых продвинутых инструментов **десятилетиями**.

30-YEAR PATENT

\$5000 TOOLS



imgflip.com

Подумайте об этом. Это не какая-то экзотика из секретных мануалов. Не недокументированная фишка нового процессора. Это базовые инструкции, описанные в патенте почти 30 лет назад. И все это время они просто... существовали. Прямо у нас под носом.

Семпл, с которого все началось, вскрыл не уязвимость в процессоре, а огромный проблему в нашем подходе.

На этом пока всё. Спасибо за прочтение.