25-ЛЕТНИЙ ПАТЕНТ INTEL, КОТОРЫЙ УБИЛ ВАШИ ДИЗАССЕМБЛЕРЫ.

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

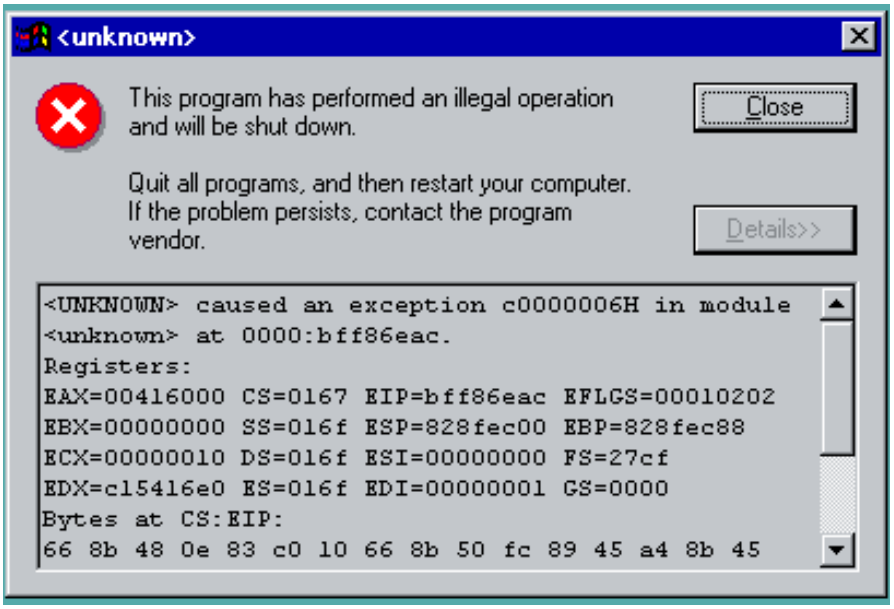
Наша история началась с сообщения от друга-реверсера colby57: *“Первый раз когда я это увидел, думал что сейчас уже улечу с крашем, но всё валидно"*. В приложении был образец шелла. Ничего особенного на первый взгляд, но посреди кода дизассемблер просто обрывался, показывая дальше "мусорные" байты. Процессор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

Этим "мусором" оказались два опкода: `0F 1A` и `0F 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 назвали **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.

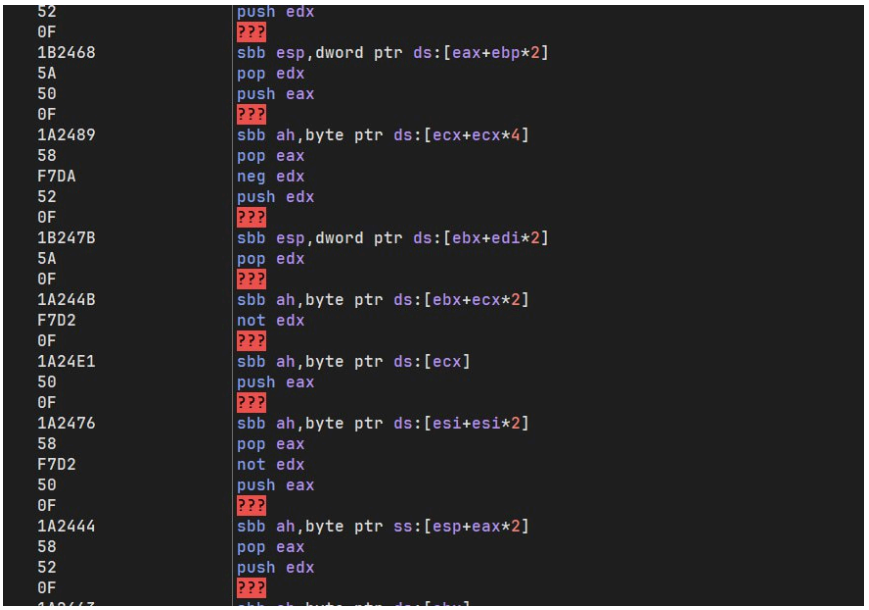
Live-тест: ломаем всё

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.

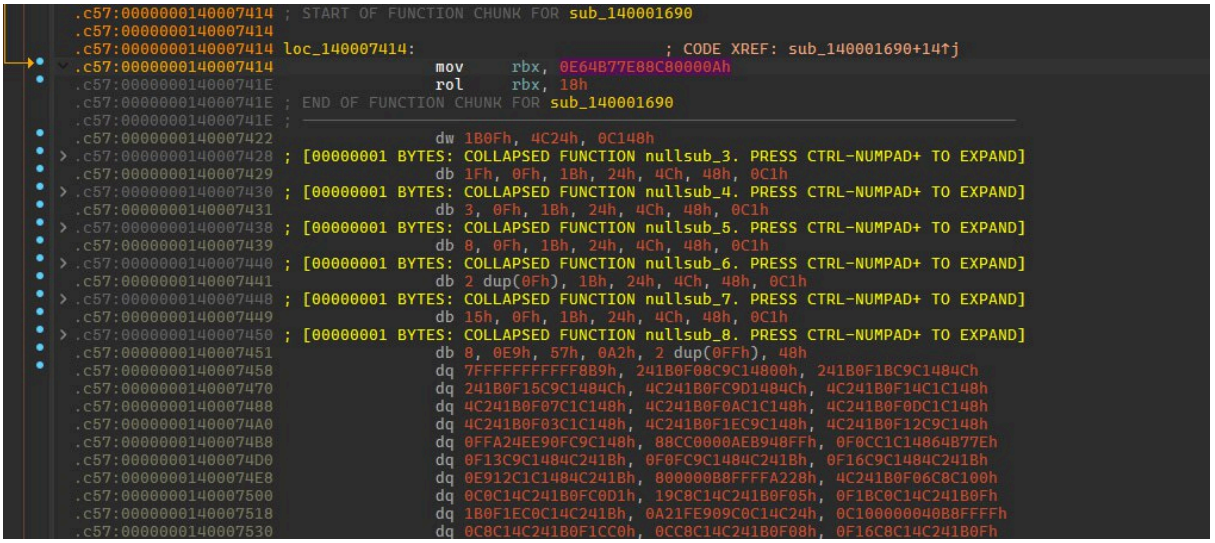


Стрелка EIP указывает на строку с байтами 0F 1A C0. В колонке дизассемблера на этой строке три вопросительных знака.

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

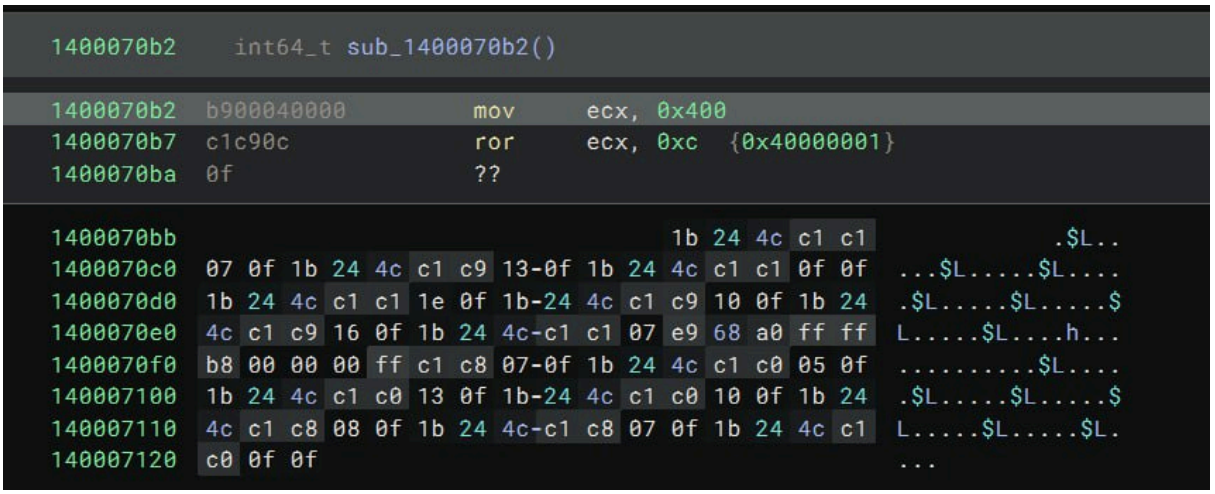
А что же скажет любимая IDA? Открываем бинарник и видим удручающую картину.



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



Как же нам это использовать?

Так что же эти два опкода дают злоумышленнику на практике?

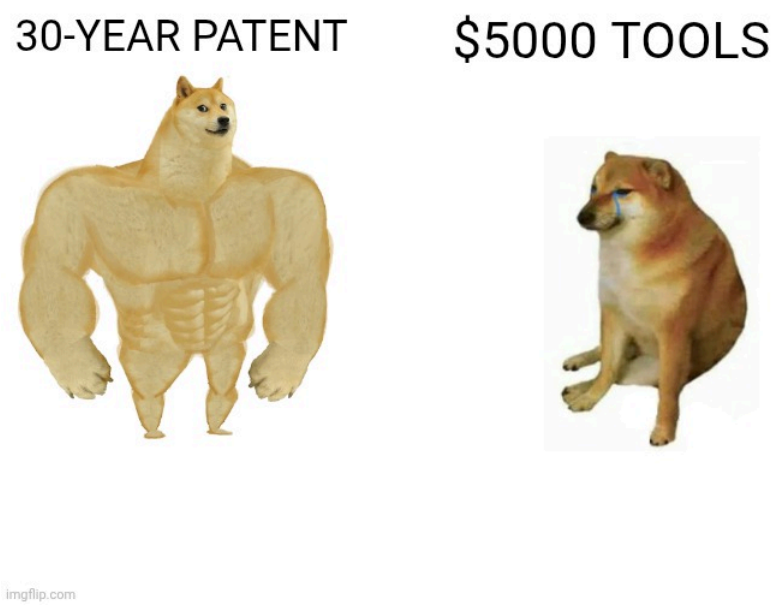
Самое очевидное — **сломать анализ**. Как мы видели в IDA, один такой NOP — и анализ функции обрывается. Весь последующий код для дизассемблера просто перестает существовать.

Пример уровня выше: **создание фейковых графов выполнения**. После `jmp` ставится комбинация `0F 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**.



Подумайте об этом. Это не какая-то экзотика из секретных мануалов. Не недокументированная фича 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 огромную проблему в нашем подходе.

На этом пока всё. Спасибо за прочтение.