Ссылки:

forcedentry citizen lab - https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/

ipwn citizen lab - https://citizenlab.ca/2020/12/the-great-ipwn-journalists-hacked-with-suspected-nso-group-imessage-zero-click-exploit/

https://www.wired.com/story/ios-security-imessage-safari/ https://citizenlab.ca/2018/09/hide-and-seek-tracking-nso-groups-pegasus-spyware-to-operations-in-45-countries/

forcedry https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html

https://googleprojectzero.blogspot.com/2022/03/forcedentry-sandbox-escape.html

https://googleprojectzero.blogspot.com/2020/04/fuzzing-imageio.html

https://googleprojectzero.blogspot.com/2021/01/a-look-at-imessage-in-ios-14.html

Состав:

Введение (2 минуты)

• Краткий обзор актуальности кибербезопасности в современном мире.

• Введение в тему эксплойта FORCEDENTRY, его значимость и влияние на область безопасности операционных систем.

Основные принципы защитных механизмов (4 минуты)

1. NX (No Execute): Объяснение принципа работы и его роли в предотвращении выполнения кода из неисполняемой памяти.

2. DEP (Data Execution Prevention): Как DEP защищает систему от атак, основанных на выполнении кода в областях данных.

3. ASLR (Address Space Layout Randomization): Роль ASLR в усложнении эксплуатации уязвимостей путём рандомизации расположения данных и кода в памяти.

Анализ уязвимостей и эксплойта FORCEDENTRY (5 минут)

1. Обзор уязвимостей: Анализ типов уязвимостей, которые эксплойт FORCEDENTRY использовал для атаки.

2. Методы эксплуатации: Как FORCEDENTRY обходит защитные механизмы, такие как NX, DEP, и ASLR.

3. Адресация уязвимостей на уровне архитектуры: Предложения по укреплению архитектурной безопасности операционных систем.

Значение песочницы для безопасности приложений (3 минуты)

1. Введение в песочницы: Объяснение концепции и основных функций песочниц.

2. BlastDoor Sandbox: Рассмотрение механизма работы и его роли в защите iMessage от вредоносного кода.

3. Побег из песочницы: Как эксплойты, подобные FORCEDENTRY, нарушают изолированную среду выполнения.

Ошибки при проектировании и их последствия (3 минуты)

1. Распространённые ошибки при проектировании: Обсуждение, как недочёты в проектировании могут создавать уязвимости.

2. Примеры из реальной жизни: Анализ ошибок, которые привели к успешной эксплуатации уязвимостей в случае FORCEDENTRY.

Заключение и рекомендации (3 минуты)

• Обобщение основных моментов доклада.

• Важность непрерывного обновления и улучшения защитных механизмов.

• Предложения по дальнейшим исследованиям и разработке в области кибербезопасности.

Введение

Современный мир все больше и больше зависит от компьютеров. Компьютеры отвечают за нашу безопасность, здоровье, финансы. Но помимо огромных преимуществ, такое внедрение цифровых систем подвергает нас риску быть взломанными. Всегда найдутся злоумышленники, которые попытаются обойти, завладеть или даже уничтожить систему ради собственной выгоды или на заказ. Наиболее популярным типом уязвимостей остаются атаки, направленные на манипулирование памятью. (картинка с графиком процента атак)

Именно про такие атаки мы и поговорим на примере эксплойта FORCEDENTRY. FORCEDENTRY (так же известный как уязвимость CVE-2021-30860) это эксплойт, разработанный Израильской компанией NSO Group (которая занимается кибербезопасностью и разработкой различного, в том числе, шпионского ПО). Суть данного эксплойта заключается в том, что из-за архитектурных ошибок в обработчике сообщений приложения iMessage злоумышленники смогли создать свой “компьютер внутри компьютера” при отправке PDF файлов зашифрованных кодеком JBIG2 под видом .GIF и, выйдя за “безопасный периметр“ (т н Sandbox), установили шпионское ПО Pegasus (от той же NSO Group). Самой интересной данного взлома особенностью, на мой взгляд, является тот факт, что FORCEDENTRY – это 0-click эксплойт, то есть, от пользователя устройства не требуется никаких действий для того, чтобы эксплойт начал свою работу.

Защитные техники

Но прежде, чем разговаривать про взлом, давайте разберем основные механизмы и подходы защиты, которые применяются при проектировании и разработке систем.

NX

NX (или No Execute) это механизм безопасности, реализованный на уровне процессора. Он помечает определенные участки памяти на данные и инструкции процессора, тем самым, делая невозможным выполнение кода на непредназначенных для этого участках. Достигается при помощи пометки области памяти, как неисполняемой при помощьи специального бита. Такой подход, в теории, защищает от буферного переполнения и выполнения произвольного кода.

В iMessage NX используется совместно с ASLR и сендбоксингом для усложнения эксплуатации различных уязвимостей. Именно этот метод должен остановить выполнение вредоносного кода, даже если злоумышленник сможет его внедрить в память.

Однако NX не способен предотвратить взлом из-за некорректной изоляции компонентов. Например, хакер сможет взломать один компонент, и благодаря этому получить контроль над другим в обход NX.

ASLR (Address Space Layout Randomization)

ASLR (Address Space Layout Randomization) — это механизм безопасности, который усложняет атаку на память путём случайного изменения адресов памяти системных приложений и данных, исполняемого кода, а также стека и кучи.

ASLR предназначен для защиты от атак, которые зависят от предсказуемости расположения кода и данных в памяти, таких как переполнение буфера и атаки внедрения кода. В случае невозможности предугадать расположение нужных данных, взлом может занять значительно большее время.

Основными ошибками при разработке ASLR являются:

• Недостаточная энтропия: если рандомизация не обеспечивает достаточно широкого диапазона адресов, атакующие могут использовать методы перебора/

• Ошибки в реализации приложений: недостаточная защищенность самого приложения (например утечки информации о памяти), могут помочь атакующим обойти ASLR, предоставляя данные, необходимые для определения расположения важных областей памяти.

DEP

DEP (Data Execution Prevention) – механизм защиты, направленный на предотвращение выполнения вредоносного кода из неиспользуемых областей памяти. Во многом схож с NX, но может быть реализован как программно, так и аппаратно. Он помечает определенные области памяти, как недоступные для выполнения кода.

В теории, DEP (как и остальные механизмы) должен ограничивать исполнение, кода, внедренного злоумышленниками. Однако, будучи некорректно сконфигурированным, не сможет предоставить нужной защиты. Например, злоумышленники могут найти процессы или компоненты, не защищенные с помощью DEP. Благодаря тому, что может быть реализован программно, является более высокоуровневым по сравнению с NX.

Сендбоксинг

Еще один подход, который часто применяется при разработке систем – это сендбоксинг. Его суть заключается в том, что каждое приложение “живет” в отдельном изолированном пространстве, что, в свою очередь, позволяет защитить операционную систему и остальные приложения от атак со стороны изолированного.

Однако неправильная конфигурация или ошибки в разработке могут привести к значительным последствиям, например замедлению работы приложений или нарушению изоляции.

Общий обзор

Общий Обзор Уязвимостей

Теперь, когда мы поговорили про основные принципы защиты, давайте перейдем к уязвимости. Как уже было сказано, FORCEDENTRY – эксплойт для iOS 14, позволявший из-за некорректной обработки медиа файлом нарушить работу операционной системе. Уязвимость была закрыта 13 сентября 2021 года с выходом iOS 14.8. FORCEDENTRY не полагается на единичную уязвимость, а использует цепочку уязвимостей для достижения конечной цели — выполнения произвольного кода на устройстве жертвы и установки шпионского ПО. Эта цепочка начинается с уязвимости в механизме обработки изображений.

Инициация через медиа файл: Атака начинается с отправки специфически созданных файлов (PDF форматированный с помощью JBIG2 и при этом помеченный, как GIF), который обрабатывается библиотекой ImageIO iOS. Эта библиотека используется для предсказания нужного формата файла (вне зависимости от расширения) и последующей обработки (чтение, преобразование, создания превью и тд) его. Стоит отметить, что трюк с подменой формата данных дает возможность использовать более 20 видов кодеков для взлома.

Небольшое отступление

Прежде чем пройти дальше, поговорим про кодировки. JBIG2 – это формат, который был создан для сканирования изображений в PDF, когда людям требовалось максимально сильно сжимать файлы из за ограничений интернета и объёма накопителей. По сути, использует 2 основах техники:  
 1. Благодаря тому, что многие языки используют сравнительно маленькие алфавиты, JBIG2 старается разделить отсканированную страницу на отдельные символы при помощи простых алгоритмов сравнения без использования оптического распознавания текста. Т е, JBIG2 не знает, что он сканирует. После этого кодировщик составляет “карту”, помечая, где какие символы должны стоять. При декомпрессии получается так, что у нас есть всего 1 образец каждой буквы или символа, из которых уже формируются файлы. Недостатком такого подхода является тот факт, что кодировщик может случайно перепутать похожие символы и тем самым изменить суть документа (например, может поменяться стоимость какого-либо изделия или реквизиты организации). Это послужило одной из причин того, что JBIG2 ушел на пенсию.

2. Так как качество отсканированных документов может страдать, был придуман еще один подход: кодировщик может хранить референсный символ (например, тот, который больше всего похож эталон, заложенный в память), а так же, для каждой буквы или символа в тексте, разницу между тем, что он отсканировал, и референсом. Во время декомпрессии, используя различные логические операторы (например XOR, AND, OR), он восстанавливает исходный символ, значительно улучшая качество изображения.

Кодировка JBIG2 состоит набора сегментов (которые нужны для расстановки символов, обозначения новой страницы и тд). В библиотеке, которая использовалась Apple, сегменты представлены классом JBIG2Segment и его подклассами (Bitmap, SymbolDict, и другими). Bitmap представляет собой простой набор символов, а SymbolDictionary обьединяет несколько Bitmap.

К сегментам можно обращаться по их номерам, либо хранить через специальный лист (Glist) указатели на конкретные сегменты.

Теперь, когда мы поговорили про принципы кодировки, мы можем рассмотреть уязвимость подробнее. В ее основе лежит простое переполнение 32-х битного целого числа.

Обратим внимание на представленный код:  
Guint numSyms = 0; // (1)

  for (i = 0; i < nRefSegs; ++i) {

    if ((seg = findSegment(refSegs[i]))) {

      if (seg->getType() == jbig2SegSymbolDict) {

        numSyms += ((JBIG2SymbolDict \*)seg)->getSize();  // (2)

      } else if (seg->getType() == jbig2SegCodeTable) {

        codeTables->append(seg);

      }

    } else {

      error(errSyntaxError, getPos(),

            "Invalid segment reference in JBIG2 text region");

      delete codeTables;

      return;

    }

  }

...

  // get the symbol bitmaps

  syms = (JBIG2Bitmap \*\*)gmallocn(numSyms, sizeof(JBIG2Bitmap \*)); // (3)

  kk = 0;

  for (i = 0; i < nRefSegs; ++i) {

    if ((seg = findSegment(refSegs[i]))) {

      if (seg->getType() == jbig2SegSymbolDict) {

        symbolDict = (JBIG2SymbolDict \*)seg;

        for (k = 0; k < symbolDict->getSize(); ++k) {

          syms[kk++] = symbolDict->getBitmap(k); // (4)

        }

      }

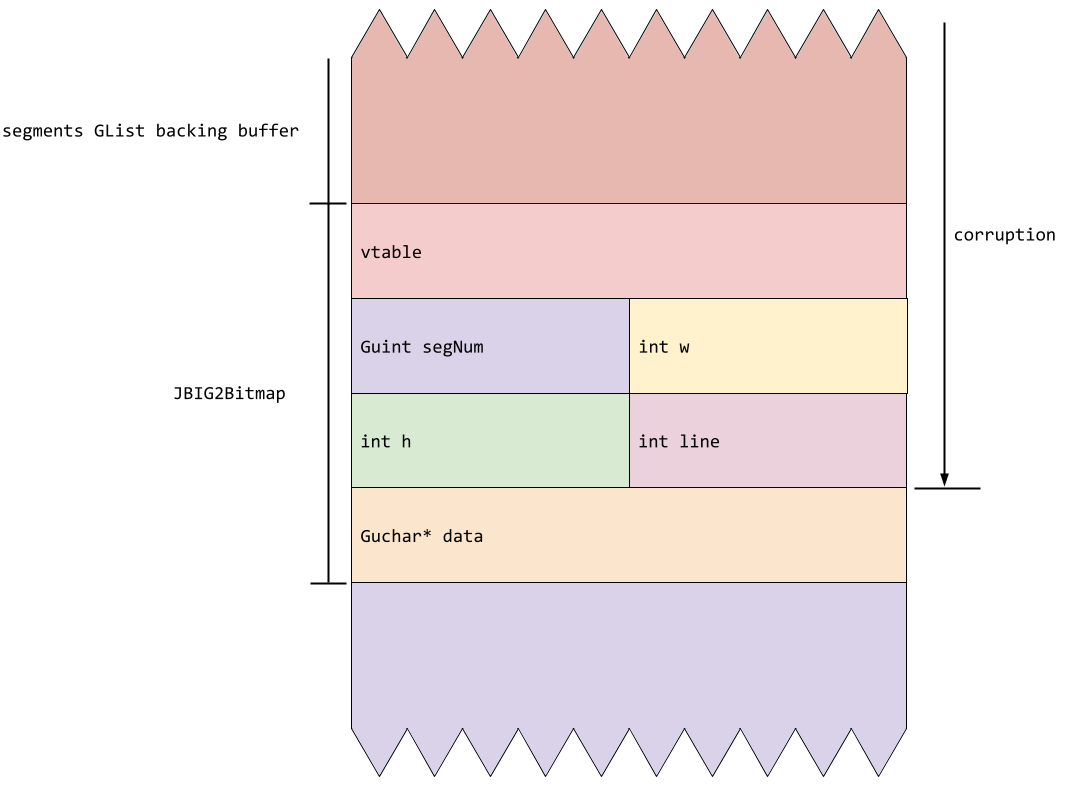
    }

  }

Суть этого кода не очень важна. Нам нужно обратить внимание на переменную numSys (1) (32-х битное целое число). Благодаря заранее подготовленному PDF файлу с кодировкой JBIG2 и отсутствию достаточной валидации значений злоумышленники добились переполнения в (2) и получили небольшое значение numSys. Это, в свою очередь, вызвало создание буфера слишком маленького размера в (3).

Такое развитие событий может привезти к записи огромного количества пустых данных, а как следствие, крашу парсера. Но логика обработки данных реализована так, что первые несколько записей мусорных данных повреждают резервный буфер Glist и перезаписывают указатели на Segment, заменяя их на Bitmap. Однако такой лавинообразный подход повредил бы все данные, не так ли? Нет. Bitmap наследует от Segment виртуальный вызов получения типа сегмента, однако Bitmap возвращает другой тип данных в этом вызове, что блокирует последующую запись и защищает злоумышленника от ненужного повреждения памяти.

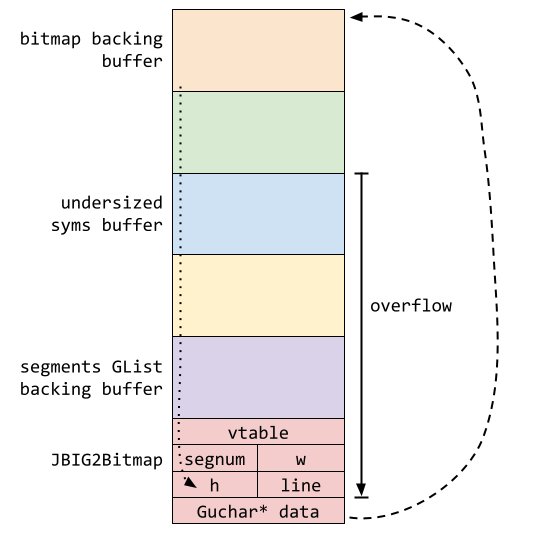
Теперь, когда Glist поврежден, начинается обработка Bitmap, который отвечает за текущую страницу. JBIG2Bitmaps — это простые оболочки вокруг резервного буфера, хранящие ширину и высоту буфера (в битах), а также значение строки, которое определяет, сколько байтов сохраняется для каждой строки.



Далее, злоумышленник переписывает еще 3 указателя Bitmap и останавливает переполнение. К этому моменту информация, отвечающая за отрисовку текущей страницы, уже будет переписана. Благодаря тому, что iOS имеет прямой порядок байтов. Из-за особенностей структуры адресного пространства iOS, указатели скорее всего будут находится в памяти с адресами между 0x100000000 и 0x1ffffffff. А так как iOS имеет little-endian порядок байтов, то поля w и line будут перезаписаны с помощью 0x1 – наиболее значимой половиной указателя JBIG2Bitmap, а segNum и h получат наименее значимую половину казателя, скорее всего довольно случайное значение.

Из-за этого, значение h у текущей страницы JBIG2Bitmap будет неизвестным, но точно очень большим. А так как h используется для проверки границ выделенной памяти и должно отображать размер выделенного буфера, мы получим расширение границ канваса. А это, в свою очередь, приведет к тому, что следующие команды смогут быть прочитаны и записаны за пределами исходных границ буфера.

Подрезание кучи (heap groom) разместит буфер текущей страницы чуть ниже undersized буфера переменной syms, что приведет к тому, что если страницы JBIG2Bitmap неограниченна, она может записывать и читать свои собственные поля с помощью команд.



Записывая специально созданные 4 байтовый битмапы в координатах канваса, злоумышленник может записывать данные во все поля JBIG2Bitmap. А подбирая значения для w, h и line, он сможет записывать данные с произвольным оффсетом относительно буфера.

Теперь злоумышленник уже сможет переписывать данные по любым адресам, если он будет знать эти адреса. Все было бы легко, если бы смогли запустить произвольный JavaScript. Но мы все еще находимся в однопроходном парсере изображений. И тут наступает самая интересная часть.

Помните, что JBIG2 умеет работать с различными логическими операторами? А что, если использовать их не для работы с символами, но и для обработки отдельных битов? Это открывает огромные возможности, ведь злоумышленник может создать целые наборы команд. А так как буфер у нас не ограничен (еще один косяк с точки зрения проектирования), эти команды могут работать с фактически произвольной памятью. Объединяя логические операции в наборы команд, злоумышленники получают возможность создать целую архитектуру и запрограммировать ее, например, для побега из песочницы.

Это была та основа, за которой стоит FORCEDENTRY. Благодаря целому набору уязвимостей: неограниченным возможностям парсера, отсутствию защиты от переполнения (и соответствующей проверки чисел), злоумышленники смогли создать целую компьютерную архитектуру и запрограммировать ее нужным образом. Все эти ошибки по отдельности не кажутся такими уж и страшными, но, если их объединить, мы получаем возможность нанести серьезный вред безопасности. А это, как я считаю, является серьезным просчетом в архитектуре модулей iOS.

Методы эксплуатации:

Прежде чем перейти к побегу из песочницы давайте обсудим работу механизмов защиты.

Технические Детали Обхода NX (No Execute)

NX, как мы помним, защищает систему, запрещая выполнение кода из областей памяти, предназначенных только для данных. Однако, эта защита оказывается неэфективной, ведь FORCEDENTRY выбирает наборы инструкций (известные как гаджеты), которые уже есть в системе; создает цепочку гаджетов и выполняет нужный код для работы уязвимости, после чего перед ним открывается возможность работы с собственными наборами инструкций.

Технические Детали Обхода ASLR (Address Space Layout Randomization)

ASLR усложняет эксплуатацию уязвимостей путём рандомизации расположения ключевых областей памяти. Несмотря на это, FORCEDENTRY находит способы обхода ASLR и манипуляции с памятью: благодаря получению контроля над памятью из-за ошибок в разработке алгоритмов и библиотек, механизм ASLR оказывается неэффективным против этого эксплойта.

Предложения

Улучшение Изоляции и Сегментации

1. Использование аппаратной изоляции: Расширить использование аппаратных средств безопасности для обеспечения дополнительных уровней изоляции критически важных процессов и данных. Это может помочь защитить чувствительные данные и операции даже в случае компрометации операционной системы.

2. Расширенная рандомизация: Улучшение и расширение принципов ASLR для включения более широкой рандомизации и увеличения энтропии расположения ключевых областей памяти, включая рандомизацию стека, кучи, и расположения библиотек в памяти.

3. Ограничение возможностей отдельных библиотек или модулей.

2. Динамический анализ поведения: Внедрение систем динамического анализа поведения программ для выявления и блокировки потенциально вредоносных действий в реальном времени, основываясь на анализе поведенческих паттернов, а не только на сигнатурах вредоносного ПО.

Песочница

Теперь давайте поговорим про ничуть не менее важный механизм защиты

Концепция Песочниц

Песочница (или же Sandbox) — это защитный механизм для отдельного запуска программ, приложений или кода в строго контролируемой среде. Эта среда ограничивает доступ к ресурсам системы, таким как файлы, сетевые соединения, системные вызовы и интерфейсы операционной системы, предотвращая тем самым возможное вредоносное воздействие на хост-систему или другие приложения.

Технические Основы

• Изоляция Процессов: Песочница использует механизмы операционной системы для изоляции процессов, такие как контейнеры или виртуальные машины. Это обеспечивает физическую и логическую изоляцию запущенного кода от остальной системы.

• Контрольный Список Ресурсов: Определяет, к каким ресурсам приложение может получить доступ. Это включает файловую систему, сетевые интерфейсы, системные вызовы и API операционной системы. Политики песочницы могут быть настроены для предоставления минимально необходимого уровня доступа.

• Ограничение Прав: Песочница ограничивает права выполнения кода, применяя принцип наименьших привилегий. Это означает, что код выполняется с минимальным набором прав, необходимых для его работы.

Примеры Применения

• Безопасность Браузеров: Современные веб-браузеры используют песочницы для изоляции вкладок и плагинов, предотвращая выполнение вредоносного кода через уязвимости в веб-контенте.

• Мобильные Операционные Системы: iOS и Android широко используют песочницы для изоляции приложений друг от друга и от системных ресурсов, уменьшая тем самым риск вредоносного воздействия на устройство.

Технические Вызовы и Ограничения

• Производительность: Изоляция и контроль доступа могут влиять на производительность приложений, особенно в средах с интенсивным использованием ресурсов.

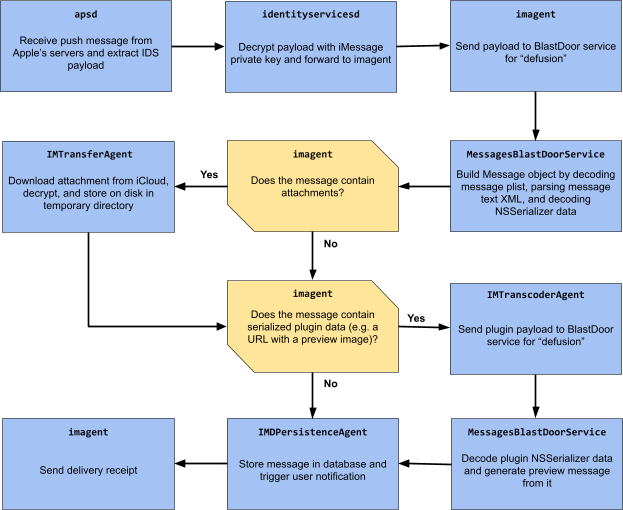
• Управление Песочницей: Настройка и управление политиками песочницы требуют глубокого понимания приложений и их потребностей в ресурсах, а также потенциальных угроз безопасности.

• Обход Механизмов Изоляции: Существуют сложные техники и эксплойты, направленные на обход механизмов изоляции песочницы, что требует постоянного анализа угроз и обновления механизмов защиты.

Теперь давайте поговорим про BlastDoor и то как он, и несколько других алгоритмов в iOS 14 должны были повлиять на безопасность устройства.

Обзор BlastDoor

BlastDoor является ключевым компонентом системы безопасности, введенным в iOS 14, специально разработанным для обработки входящего контента iMessage. Основа BlastDoor создана на мемори сейв языке Swift. Этот механизм представляет собой изолированную среду, предназначенную для безопасной обработки и анализа данных полученных из вне сообщений, включая текст, изображения, видео и другие вложения, прежде чем они будут представлены пользователю. Цель BlastDoor — предотвратить выполнение вредоносного кода, эксплуатирующего уязвимости в системе обработки сообщений.



Защита iMessage

BlastDoor играет центральную роль в стратегии защиты iMessage, обеспечивая несколько уровней защиты. BlastDoor тщательно десериализует входящие сообщения, преобразуя их из формата, используемого для сетевой передачи, в формат, пригодный для обработки и отображения. Этот процесс минимизирует риски, связанные с обработкой ненадежных или специально сформированных данных.

Обновленный сендбоксинг в новой версии iOS допускает использование лишь незначительное число действий:

В песочнице приложение имеет доступ к ограниченному числу системных модулей, отвечающих за IPC, namely diagnosticd, logd, opendirectoryd, syslogd, and notifyd, can be reached

Серьезные ограничения по доступу к файловой системе

Запрет на взаимодействие с драйверами IOKit (historically a big source of vulnerabilities) is forbidden

Значительно ограничен исходящий доступ к сети

Еще одним важным введением стало исправление ASLR: до обновления iOS 14 у Apple была серьезная архитектурная проблема, связанная с ним. Большинство библиотек, используемых системой, были доступны заранее, причем адрес рандомизировался только один раз, при запуске системы. Теперь же, когда какой-то сервис пытается получить доступ к кешу библиотек, адрес этого участка памяти рандомизируется каждый раз.

Так же, для усложнения взломов, рассчитанных на перебор или работу с крашами приложений был введен троттлинг этих приложений. Теперь скорость работы каждого приложения или сервиса после его краша намеренно занижается, что значительно увеличивает время, требуемое на валом (вместо пары десятков секунд до нескольких десятков часов).

Концепция "Побега из Песочницы"

Теперь, когда мы знаем про нововведения в безопасности iOS 14, поговорим про то, как FORCEDENTRY их обошел. Итак, злоумышленник уже смог создать “компьютер” внутри песочницы. Следующая цель – это выйти за пределы изолированной зоны и установить уже настоящее шпионское ПО.

Для начала стоит поговорить о тех вещах, которые использовались для побега. В этом взломе хакеры использовали регулярные выражения и предикаты для достижения своих целей (NSExpression and NSPredicate). Благодаря тому, что поле isa у обьектов в языке Objective C (расписать про поле) не было защищено кодами аутентификации указателей (PAC, про них тоже насрать), компьютер, созданный в парсере, имел возможность создавать поддельные обьекты Objective C, которые в свою очередь вызывали dealloc. Это приводило к тому, что выражение NSFunctionExpression десериализовывались и “инициализировались”.

На прошлом шаге злоумышленники создали набор NSFunctionExpression. Этот обьект нужен для двух вещей: 1) замести следы, удаляя те самые гифки, из-за которых все началось; 2) создание пейлода и получение Pegasus’а.

Поговорим про заметание следов. Шпионское ПО на то и шпионское, чтобы остаться незамеченным. Функциональное выражение вызывает файловый менеджер системы, выбирает из временных файлов гифки, которые были использованны для взлома и удаляет их. Просто и элегантно.

Теперь идем дальше. Для вызова методов из других процессов в Objective C используется такой механизм, как NSXPConnection (semitransparent rpc mechanism). Важно отметить, что для “пересылки” объектов через этот механизм необходимо, чтобы они имели свои протоколы, описывающие доступные для вызова методы. Для сериализации доступны подклассы любых классов без ограничения на наследственность. Это означает, что любой объект протокола, который объявляет определенный тип поля, так же принимает любой подкласс этого типа. Таким образом, протокол допускает использование любого подкласса (а это уже является критической угрозой безопасности).

Теперь можно передать Array состоящий из двух элементов: объекта синтеза голоса (странно, да?) и объекта PTSection с одним рядом PTRow. Но ведь для использования прототипа нам наверняка нужны соответствующие библиотеки. Для этого нам и нужен синтез голоса. Когда мы передаем объект синтеза голоса, рантайм Objective C вызывает метод initialize, что в свою очередь подгружает ряд библиотек для него. Одной из них и является библиотека для работы с PrototypeTool.

Что же тогда нас ждет в PTRow? Старый добрый предикат. Его десериализация и присвоение PTRow вызовет allowEvaluation. Таким образом, операционная система не будет выполнять никаких проверок предиката, что является, по сути, доступом к выполнению произвольного кода (еще одна критическая ошибка!).

Такой же подход используется для “выполнения” еще несколько пейлодов, но уже в контексте модуля, отвечающего за связь. Они делают следующее:

1. Создания URL по определенному адресу и сохранение его
2. Подгрузка дополнительных библиотек (так же, как с синтезом голоса)
3. Параметризация URL из первого шага с указанием точной модели и других параметров телефона
4. Создание полного URL адреса и получение данных сервера, и расшифровка

Таким образом, злоумышленник смог уже сбежать из сендбокса и запросить следующую информацию с управляющего сервера. В этот момент уже становится ясно, что хакеры нанесли серьезный ущерб с точки зрения безопасности, воспользовавшись большим количеством ошибок в архитектуре.

Заключение

Итак, главной особенностью побега является не какие то специальные методики по взлому, а банальное использование дыр в ПО.

И все это работает не на какой то специфичной версии ПО; оно даже никак не повреждает память, как будто так и должно было все работать.

На примере эксплойта FORCEDENTRY мы познакомились с такими КРИТИЧЕСКИМИ ошибками при разработке, как:

Недостатки в Обработке Входных Данных

FORCEDENTRY использовал специально сформированные файлы, чтобы эксплуатировать уязвимости в обработке изображений и других медиа-форматов внутри iMessage. Эти файлы были разработаны так, чтобы вызвать непредвиденное поведение в программном обеспечении, что часто является результатом:

Переполнения Буфера: Когда данные превышают выделенный для них буфер, что может привести к перезаписи важных участков памяти.

Ошибки Десериализации: Неправильная обработка входных данных может привести к выполнению произвольного кода при десериализации объектов.

Использование Логических Ошибок

Атака использовала логические ошибки в программном обеспечении, такие как недостаточная проверка состояний или условий, под которыми могут быть обработаны данные, приводя к несанкционированному выполнению кода.

Обход Механизмов Защиты

Обход ASLR и DEP: FORCEDENTRY демонстрирует, как атакующие могут использовать сложные методы для обхода механизмов защиты, таких как Address Space Layout Randomization (ASLR)

Недостающая изоляция. Возможность определенных модулей взаимодействовать с теми, которые джае в теории им не будут нужны, а так же доступ к различным специфичным библиотекам могут привести к нарушению безопасности.

Данный эксплоит продемонстрировал, что ошибки, которые выглядят мелко и незначительно по отдельности, способны нанести серьезный ущерб и скомпрометировать такое сложное устройство, как смартфон. Хакеры собрали из них сложную систему, которая смогла не только собрать полноценный компьютер в буфере, но и выйти из сендбокса и скачать шпионское ПО.

Помимо этих, распространенными являются следующие ошибки:

1. Недостаточная валидация входных данных (например sql иньекции, xss атаки)
2. Ошибки управления памятью. Например, Use-After-Free: обращение к памяти после того как она освобождена может привести к выполнению вредоносного кода, если хакер имеет доступ к ней.
3. Недостаточная аутентификация: слабые политики аутентификации могут позволить получить доступ к учетным данным пользователей. Из за проблем с авторизацией пользователи могут получить доступ к фунциям, которые должны быть недоступны для них
4. Неправильная обработка ошибок: если приложение возвращает подробные сообщения об ошибках пользователю, это может предоставить атакующим ценную информацию о внутреннем устройстве системы.
5. Отказ в обслуживании: Неправильная обработка исключений может привести к сбоям в работе приложения, что может быть использовано для атак типа отказ в обслуживании (DoS).

Заключение

Случай FORCEDENTRY подчеркивает важность комплексного подхода к безопасности программного обеспечения, включая тщательное проектирование, регулярное тестирование и быстрое реагирование на обнаруженные уязвимости. Это демонстрирует, что даже в высокозащищенных системах могут существовать сложные уязвимости, которые могут нанести серьезный ущерб безопасности.