Т. Р. Хабибуллин, А. Г. Бельтов, И. Ю. Жуков, А. В. Зуйков, А. С. Смирнов

УЯЗВИМОСТИ ПО ДЛЯ МОБИЛЬНЫХ ТЕЛЕФОНОВ И БЕЗОПАСНЫЕ МЕТОДЫ ПРОГРАММИРОВАНИЯ

Современные мобильные телефоны предоставляют все больше возможностей своим владельцам. С развитием мобильных технологий растет количество и сложность программного обеспечения, устанавливаемого на смартфоны и коммуникаторы. С увеличением количества строк кода растет как число ошибок, так и их сложность, а также то воздействие, которое они оказывают на логику работы приложения. Если в относительно простой программе ошибки вызывают незапланированное завершение ее выполнения или появление сообщений, то недостатки более сложных могут привести к утечке памяти, порче данных и, самое опасное, появлению уязвимостей.

Наиболее распространенные цели злоумышленников — внедрение и выполнение собственного кода (который в этом случае называется shell-кодом), получение доступа к данным, которыми оперирует приложение, искажение информации в пространстве памяти программы и т. д.

Все перечисленные угрозы, в особенности исполнение собственного кода на атакованной платформе, являются тем более опасными, если приложение запущено от имени суперпользователя (администратора). В этом случае shell-код может получить доступ к любым файлам, находящимся на устройстве, выполнить переконфигурацию системы, создать и запустить троянское приложение, которое в таком случае тоже будет выполняться с правами администратора, получить любые данные о пользователе.

Приложение, в котором обнаружена уязвимость, является таким же способом доступа в систему, как и терминал, с той лишь разницей, что для его использования требуется некоторый профессиональный навык. Последнее, впрочем, утрачивает актуальность с развитием сети Интернет и появлением хакеров, создающих скрипты для своих менее способных коллег.

Перейдем теперь к рассмотрению конкретных методов и техник, которые используются при атаке.

Первыми в списке по популярности среди хакеров, частоте появления в программах и серьезности угрозы, которую они представляют, являются ошибки, связанные с переполнением буфера [1].

Такие ошибки, с одной стороны, представляют собой общеизвестную потенциальную уязвимость любого приложения, позволяющую проводить различные атаки: DoS, внедрение shell-кода, получение доступа к секретным переменным, а с другой стороны, в случае таких языков, как C, всегда можно найти процедуру, где реализация алгоритма возможна только с учетом доверия к вызывающему коду. Приведем простой пример — реализация любой обработки строк.

Листинг 1. Простейшая реализация обработчика строк

```
void some_string_parsing(char* pstring)
{
 while (*pstring)
 {
        //обработка
              ++pstring;
 }
}
```

Т. е. если мы не обнаружим завершающего нуля, который должен присутствовать в конце строки, то пойдем по ОЗУ, пока pstring не укажет на 0 либо пока не произойдет ошибки страничной адресации (если устройство реализует страничную организацию памяти) или обращения за предел диапазона памяти.

Таким образом, при программировании на ряде языков (к которым относится большая часть языков — наследников C) будут возникать ситуации, требующие доверия к поступающей информации. Задача разработчика в данном случае — отсечь непроверенные данные от кода, производящего заключительную обработку, посредством цикла проверок.

В зависимости от функции и способа формирования буфера (находится он в стеке, куче или сегменте данных) приложение может получить несанкционированный доступ к памяти или даже изменить те ее области, на модификацию которых у него нет прав.

Все подобные ошибки можно классифицировать по типу переполняемого буфера — в стеке (автоматический), в куче (динамический), в сегменте данных (статический) [2].

Второй независимой классификацией уязвимостей может быть способ доступа к переполняемому буферу — последовательный либо индексный.

В первом случае имеется в виду переполнение, вызывающее обращение к памяти на чтение/запись последовательно ко всем ячейкам за пределами диапазона буфера. К такой ситуации может привести использование кода, описанного в листинге 1.

Индексное переполнение возникает в случае обращения к ячейке либо к диапазону за пределами буфера по индексу (т. е. потенциально мы можем обратиться как к адресам, большим базового адреса буфера, так и к меньшим). Пример в листинге 2 демонстрирует уязвимую реализацию получения кода символа.

Листинг 2. Индексное переполнение

```
long get_long(char* cbuff, int i)
{
 long retval = *(long*)(cbuff + sizeof(long)*i);
 return retval;
}
```

Наконец, третья независимая классификация — по способу доступа к памяти при переполнении (будет это запись или чтение). В первом случае важно знать, является ли запись полностью определенной алгоритмом (например, замена одних служебных символов другими), или информацию можно контролировать (например, при переписывании буферов).

Приведенные способы классификации дают возможность в каждом конкретном случае создать модель поведения злоумышленника и определить ситуацию возникновения переполнения, а также к чему такие ошибки могут привести.

Так, автоматические буферы позволяют модифицировать адрес возврата (что, в свою очередь, становится мощным средством внедрения исполняемого кода) и получить доступ к локальным переменным.

Переполнение статических буферов — механизм доступа к сегменту данных, где может храниться такая информация, как пароли и ключи, которые использует приложение.

Наиболее сложным для атаки (и практически не используемым) является переполнение динамических буферов. В данном случае все зависит от механизма, по которому происходит выделение памяти (так называемой памяти в «куче») и который меняется в зависимости от компилятора и операционной системы.

Пример реально встречающейся уязвимости, основанной на некорректной обработке входных данных, приводящей к переполнению, описан в листинге 3.

Листинг 3. Некорректный ввод, приводящий к переполнению

```
char buff[40];
get_packet(buff, 40);
short slen = *(short*)buff;                    //считывание длины буфера
short cur = 1;
short sblen;                                   //считывание длины блока данных
do{                                            //обработка пакета
switch(buff[cur++]){
case 0:                              //один из возможных кодов
sblen = *(short*)buff[cur++];              //чтение длины блока данных
parse_packet_0(buff + cur, sblen); //обработка данных указанной длины — длина блока данных считывается из полученного пакета и не проверяется, что может привести к переполнению буфера в процедуре parse_packet
cur += sblen;
break;
default:
return;
}
}while(cur < slen);
```

Другой распространенной ошибкой, приводящей к появлению уязвимости, является TOCTOU (Time Of Check — Time Of Use, время проверки — время использования). Она относится к достаточно обширному классу ошибок синхронизации и заключается в следующем: между проверкой ресурса и его использованием всегда есть определенный промежуток времени, за который может произойти либо потеря доступа к ресурсу, либо его подмена.

Пусть, например, обмен сообщениями с сервером происходит в незашифрованном виде (для экономии ресурсов), однако перед началом обмена проводится его аутентификация. В этом случае злоумышленник получает возможность перехватывать данные, но не может модифицировать их. Тем не менее если произвести переключение канала связи после подтверждения подлинности, можно получить уже полный контроль над потоком информации. В такой ситуации переходное незащищенное состояние начинается после прохождения аутентификации и продолжается либо до отключения клиентской машины, либо до проведения повторной проверки подлинности.

Приведем пример еще одной уязвимости, целиком лежащей на совести программиста, уязвимости, которую сложно встретить, но очень просто использовать. Это непреднамеренные switch fallthroughs. Практически в любом языке есть конструкции вида:

```
switch(переменная)
case значение1
действие1
break
case значение2
действие2
……
```

В случае отсутствия завершающего оператора break в блоке case начнет выполняться блок, следующий за ним по порядку. К чему это может привести — целиком зависит от тех действий, которые следующий блок выполняет.

Уязвимости, возникающие в приложении при обнаружении в нем ошибки, представляют опасность даже в том случае, когда приложения выполняются в изолированной среде (так называемой «песочнице»). В этом случае взломщик получит доступ к тем возможностям, которые были даны приложению в рамках его «песочницы». Платформы, построенные на принципе «все приложения — в песочнице», такие как Google Android, предоставляют выполняемым программам достаточно обширные привилегии по доступу к устройству, пытаясь найти баланс между поддержанием безопасности и сохранением функциональности.

Набор правил и методов, позволяющих избежать перечисленных ошибок, называется безопасным программированием. Его основные концепции — проверка данных перед обработкой, работа с буферами с обязательным контролем выхода за его границы, не привязанным к структуре информации в нем, объединение процедуры проверки и использования ресурса в атомарную операцию.

На сегодняшний день большая часть атак на мобильные устройства совершается с использованием уязвимостей в конкретном программном обеспечении.

Единственным средством, которое реально способно защитить данные пользователя, операционную систему и само приложение от ошибок в нем, является создание внутрисистемных архитектурных решений, таких как мандатный режим доступа к данным (как на ПЗУ, так и в ОЗУ), защита от модификации исполняемого кода до и во время выполнения, мандатный и монополизируемый доступ к ресурсам.

Перечисленных механизмов на сегодняшний день нет ни в операционных системах для десктопов, ни в мобильных ОС, что приводит к ежедневному увеличению армии вирусов, троянов и эксплойтов, позволяющих с минимальными усилиями выполнить взлом.

СПИСОК ЛИТЕРАТУРЫ:

1. *Касперски К.* Техника и философия хакерских атак. М.: СОЛОН-Р, 2004. — 272 с.
2. *Роббинс Дж.* Отладка приложений. СПб.: БХВ-Петербург, 2001. — 512 с.