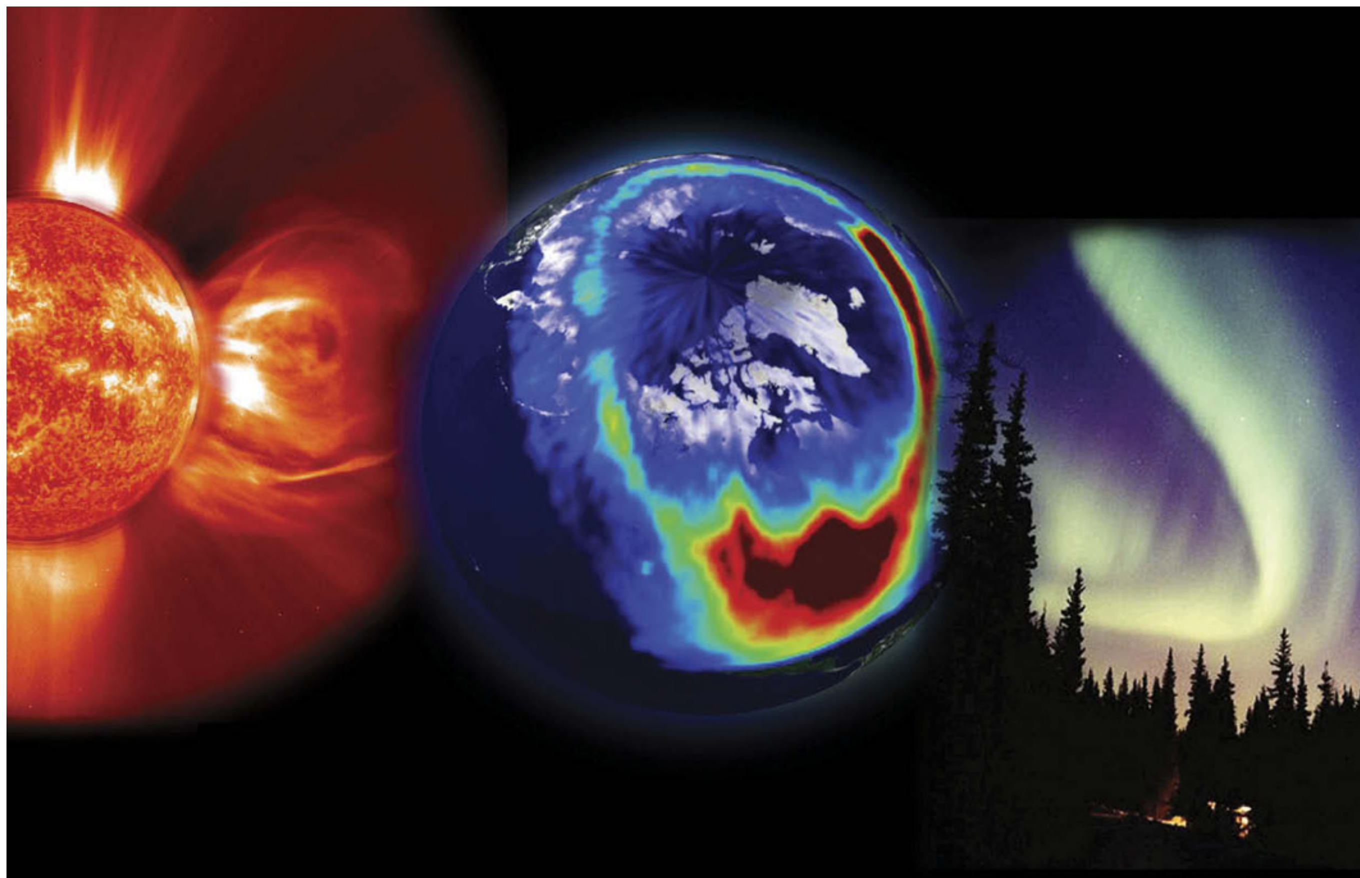
ОШИБКИ ПЕРЕПОЛНЕНИЯ БУФЕРА ИЗВНЕ И ИЗНУТРИ КАК ОБОБЩЕННЫЙ ОПЫТ РЕАЛЬНЫХ АТАК



Мы живем в суровом мире. Программное обеспечение, окружающее нас, содержит дыры, многие из которых размерами со слона. В дыры лезут хакеры, вирусы и черви, совершающие набеги изо всех концов Сети. Червям противостоят антивирусы, заплатки, брандмауэры и другие умные слова, существующие лишь на бумаге и бессильные сдержать размножение червей в реальной жизни. Сеть небезопасна – это факт. Можно до потери пульса закидывать Била Гейтса тухлыми яйцами и кремовыми тортами, но ситуация от этого вряд ли изменится.

Анализ показывает, что подавляющее большинство червей и хакерских атак основано на ошибках переполнения буфера, которые носят фундаментальный характер и которых не избежало практически ни одно полновесное приложение. Попытка разобраться в этой, на первый взгляд довольно скучной и незатейливой проблеме безопасности погружает вас в удивительный мир, полный приключений и интриг. Захват управления системой путем переполнения буфера – сложная инженерная задача, требующая нетривиального мышления и превосходной экипировки. Диверсионный код, заброшенный на выполнение, находится в весьма жестких и агрессивных условиях, не обеспечивающих и минимального уровня жизнедеятельности.

Если вам нужен путеводитель по стране переполняющихся буферов, снабженный исчерпывающим руководством по выживанию, – эта статья для вас!

КРИС КАСПЕРСКИ

Чудовищная сложность современных компьютерных систем неизбежно приводит к ошибкам проектирования и реализации, многие из которых позволяют злоумышленнику захватывать контроль над удаленным узлом или делать с ним что-то нехорошее. Такие ошибки называются дырами или уязвимостями (holes и vulnerability соответственно).

Мир дыр чрезвычайно многолик и разнообразен: это и отладочные люки, и слабые механизмы аутентификации, и функционально-избыточная интерпретация пользовательского ввода, и некорректная проверка аргументов и т. д. Классификация дыр чрезвычайно размыта, взаимно противоречива и затруднена (во всяком случае, своего Карла Линнея дыры еще ждут), а методики их поиска и «эксплуатации» не поддаются обобщению и требуют творческого подхода к каждому отдельному случаю. Было бы наивно надеяться, что одна-единственная публикация сможет описать весь этот зоопарк! Давайте лучше сосредоточимся на ошибках переполнения буферов как на наиболее важном, популярном, перспективном и приоритетном направлении.

Большую часть статьи мы будем витать в бумажных абстракциях теоретических построений и лишь к концу спустимся на ассемблерную землю, обсуждая наиболее актуальные проблемы практических реализаций. Нет, не подумайте! Никто не собирается в сотый раз объяснять, что такое стек, адреса памяти и откуда они растут! Эта публикация рассчитана на хорошо подготовленную читательскую аудиторию, знающую ассемблер и бегло изъясняющуюся на Си/Си++ без словаря. Как происходит переполнение буфера, вы уже представляете и теперь хотели бы ознакомиться с полным списком возможностей, предоставляемых переполняющимися буферами. Какие цели может преследовать атакующий? По какому принципу происходит отбор наиболее предпочтительных объектов атаки?

Другими словами, сначала мы будем долго говорить о том, что можно сделать с помощью переполнения, и лишь потом перейдем к вопросу «как именно это сделать». В любом случае эта тема заслуживает отдельной статьи.

Описанные здесь приемы работоспособны на большинстве процессорных архитектур и операционных систем (например, UNIX/SPARC). Пусть вас не смущает, что приводимые примеры в основном относятся к Windows NT и производным от нее системам. Просто в момент написания статьи другой операционной системы не оказалось под рукой.

Мясной рулет ошибок переполнения, или попытка классификации

Согласно «Новому словарю хакера» Эрика Раймонда ошибки переполнения буфера – это «то, что с неизбежностью происходит при попытке засунуть в буфер больше, чем тот может переварить». На самом деле, это всего лишь частный случай последовательного переполнения при записи. Помимо него существует индексное переполнение, заключающееся в доступе к произвольной ячейке памяти за концом буфера, где под «доступом» понимаются как операции чтения, так и операции записи.

Переполнение при записи приводит к затиранию, а следовательно, искажению одной или нескольких переменных (включая служебные переменные, внедряемые компилятором, такие, например, как адреса возврата или указатели this), нарушая тем самым нормальный ход выполнения программы и вызывая одно из следующих последствий:

- нет никаких последствий;
- программа выдает неверные данные или, попросту говоря, делает из чисел винегрет;
- программа «вылетает», зависает или аварийно завершается с сообщением об ошибке;
- программа изменяет логику своего поведения, выполняя незапланированные действия.

Переполнение при чтении менее опасно, т.к. «всего лишь» приводит к потенциальной возможности доступа к конфиденциальным данным (например, паролям или идентификаторам TCP/IP-соединения).

```
Листинг 1. Пример последовательного переполнения буфера при записи

seq_write(char *p)
{
    char buff[8];
    …
    strcpy(buff, p);
}

Листинг 2. Пример индексного переполнения буфера при чтении

idx_write(int i)
{
    char buff[]="0123456789";
    …
    return buff[i];
}
```

За концом буфера могут находиться данные следующих типов: другие буфера, скалярные переменные и указатели или же вовсе может не находиться ничего (например, невыделенная страница памяти). Теоретически за концом буфера может располагаться исполняемый код, но на практике такая ситуация почти никогда не встречается.

Наибольшую угрозу для безопасности системы представляют именно указатели, поскольку они позволяют атакующему осуществлять запись в произвольные ячейки памяти или передавать управление по произвольным адресам, например, на начало самого переполняющегося буфера, в котором расположен машинный код, специально подготовленный злоумышленником и обычно называемый shell-кодом.

Буфера, располагающиеся за концом переполняющегося буфера, могут хранить некоторую конфиденциальную информацию (например, пароли). Раскрытие чужих паролей, равно как и навязывание атакуемой программе своего пароля – вполне типичное поведение для атакующего.

Скалярные переменные могут хранить индексы (и тогда они фактически приравниваются к указателям), флаги, определяющие логику поведения программы (в том числе и отладочные люки, оставленные разработчиком), и прочую информацию.

В зависимости от своего местоположения буфера делятся на три независимые категории:

- локальные буфера, расположенные в стеке и часто называемые автоматическими переменными;
- статичные буфера, расположенные в секции (сегменте) данных;
- динамические буфера, расположенные в куче. Все они имеют свои специфичные особенности переполнения, которые мы обязательно рассмотрим во всех подробностях, но сначала немного пофилософствуем.

Неизбежность ошибок переполнения в исторической перспективе

Ошибки переполнения – это фундаментальные программистские ошибки, которые чрезвычайно трудно отслеживать и фундаментальность которых обеспечивается самой природой языка Си – наиболее популярного языка программирования всех времен и народов, – а точнее его низкоуровневым характером взаимодействия с памятью. Поддержка массивов реализована лишь частично, и работа с ними требует чрезвычайной аккуратности и внимания со стороны программиста. Средства автоматического контроля выхода за границы отсутствуют, возможность определения количества элементов массива по указателю и не ночевала, строки, завершающиеся нулем, – вообще песня…

Дело даже не в том, что малейшая небрежность и забытая или некорректно реализованная проверка корректности аргументов приводит к потенциальной уязвимости программы. Корректную проверку аргументов невозможно осуществить в принципе! Рассмотрим функцию, определяющую длину переданной ей строки и посимвольно читающую эту строку до встречи с завершающим ее нулем. А если завершающего нуля на конце не окажется? Тогда функция вылетит за пределы утвержденного блока памяти и пойдет чесать непаханую целину посторонней оперативной памяти! В лучшем случае это закончится выбросом исключения. В худшем – доступом к конфиденциальным данным. Можно, конечно, передать максимальную длину строкового буфера с отдельным аргументом, но кто поручится, что она верна? Ведь этот аргумент приходится формировать вручную, и, следовательно, он не застрахован от ошибок. Короче говоря, вызываемой функции ничего не остается, как закладываться на корректность переданных ей аргументов, а раз так – о каких проверках мы вообще говорим?!

С другой стороны – выделение буфера возможно лишь после вычисления длины принимаемой структуры данных, т.е. должно осуществляться динамически. Это препятствует размещению буферов в стеке, поскольку стековые буфера имеют фиксированный размер, задаваемый еще на стадии компиляции. Зато стековые буфера автоматически освобождаются при выходе из функции, снимая это бремя с плеч программиста и предотвращая потенциальные проблемы с утечками памяти. Динамические буфера, выделяемые из кучи, намного менее популярны, поскольку их использование уродует структуру программы. Если раньше обработка текущих ошибок сводилась к немедленному return, то теперь перед выходом из функции приходится выполнять специальный код, освобождающий все, что программист успел к этому времени понавыделять. Без критикуемого goto эта задача решается только глубоко вложенными if, обработчиками структурных исключений, макросами или внешними функциями, что захламляет листинг и служит источником многочисленных и трудноуловимых ошибок.

Многие библиотечные функции (например, gets, sprintf) не имеют никаких средств ограничения длины возвращаемых данных и легко вызывают ошибки переполнения. Руководства по безопасности буквально кишат категорическими запретами на использование последних, рекомендуя их «безопасные» аналоги – fgets и snprintf, явно специфицирующие предельно допустимую длину буфера, передаваемую в специальном аргументе. Помимо неоправданного загромождения листинга посторонними аргументами и естественных проблем с их синхронизацией (при работе со сложными структурами данных, когда один-единственный буфер хранит много всякой всячины, вычисление длины оставшегося «хвоста» становится не такой уж очевидной арифметической задачей, и здесь очень легко допустить грубые ошибки) программист сталкивается с необходимостью контроля целостности обрабатываемых данных. Как минимум необходимо убедиться, что данные не были варварски обрезаны и/или усечены, а как максимум – корректно обработать ситуацию с обрезанием. А что мы, собственно, здесь можем сделать? Увеличить буфер и повторно вызывать функцию, чтобы скопировать туда остаток? Не слишком-то элегантное решение, к тому же всегда существует вероятность потерять завершающий нуль на конце.

В Си++ ситуация с переполнением обстоит намного лучше, хотя проблем все равно хватает. Поддержка динамических массивов и «прозрачных» текстовых строк наконец-то появилась (и это очень хорошо), но подавляющее большинство реализаций динамических массивов работает крайне медленно, а строки тормозят еще сильнее, поэтому в критических участках кода от них лучше сразу же отказаться. Иначе и быть не может, поскольку существует только один способ построения динамических массивов переменной длины – представление их содержимого в виде ссылочной структуры (например, двунаправленного списка). Для быстрого доступа к произвольному элементу список нужно индексировать, а саму таблицу индексов где-то хранить. Таким образом, чтение/запись одного-единственненного символа выливается в десятки машинных команд и множество обращений к памяти (а память была, есть и продолжает оставаться самым узким местом, существенно снижающим общую производительность системы).

Даже если компилятор вдруг решит заняться контролем границ массива (одно дополнительное обращение к памяти и три-четыре машинных команды), это не решит проблемы, поскольку при обнаружении переполнения откомпилированная программа не сможет сделать ничего умнее, чем аварийно завершить свое выполнение. Вызов исключения не предлагается, поскольку если программист забудет его обработать (а он наверняка забу-

дет это сделать), мы получим атаку типа отказ в обслуживании. Конечно, это не захват системы, но все равно нехорошо.

Так что ошибки переполнения были, есть и будут! От этого никуда не уйти, и коль скоро мы обречены на длительное сосуществование с последними, будет нелишним познакомиться с ними поближе…

Окутанные желтым туманом мифов и легенд

Журналисты, пишущие о компьютерной безопасности, и эксперты по безопасности, зарабатывающие на жизнь установкой этих самых систем безопасности, склонны преувеличивать значимость и могущество атак, основанных на переполнении буфера. Дескать, хакеры буфера гребут лопатой, и если не принять адекватных (и весьма дорогостоящих!) защитных мер, ваша информация превратится в пепел.

Все это так (ведь и на улицу лишний раз лучше не выходить – случается, что и балконы падают), но за всю историю существования компьютерной индустрии не насчитывается и десятка случаев широкомасштабного использования переполняющихся буферов для распространения вирусов или атак. Отчасти потому, что атаки настоящих профессионалов происходят бесшумно. Отчасти – потому, что настоящих профессионалов среди современных хакеров практически совсем не осталось…

Наличие одного или нескольких переполняющихся буферов еще ни о чем не говорит, и большинство ошибок переполнения не позволяет атакующему продвинуться дальше банального DoS. Вот неполный перечень ограничений, с которыми приходится сталкиваться червям и хакерам:

- строковые переполняющиеся буфера (а таковых среди них большинство) не позволяют внедрять символ нуля в середину буфера и препятствуют вводу некоторых символов, которые программа интерпретирует особым образом;
- размер переполняющихся буферов обычно оказывается катастрофически мал для вмещения в них даже простейшего загрузчика или затирания сколь-нибудь значимых структур данных;
- абсолютный адрес переполняющегося буфера атакующему чаще всего неизвестен, поэтому приходится оперировать относительными адресами, что очень непросто с технической точки зрения;
- адреса системных и библиотечных функций меняются от одной операционной системы к другой – это раз. Ни на какие адреса уязвимой программы также нельзя закладываться, поскольку они непостоянны (особенно это актуально для UNIX-приложений, компилируемых каждым пользователем самостоятельно) – а это два;
- наконец, от атакующего требуется глубокое знание команд процессора, архитектуры операционной системы, особенностей различных компиляторов языка, свободное от академических предрассудков мышление, плюс уйма свободного времени для анализа, проектирования и отладки shell-кода.

А теперь для контраста перечислим мифы противоположной стороны – стороны защитников информации, с какой-то детской наивностью свято верящих, что от хакеров можно защититься хотя бы в принципе:

- не существуют никаких надежных методик автоматического (или хотя бы полуавтоматического) поиска переполняющихся буферов, дающих удовлетворительный результат, и по-настоящему крупные дыры не обнаруживаются целенаправленным поиском. Их открытие – игра слепого случая;
- все разработанные методики борьбы с переполняющимися буферами снижают производительность (под час очень значительно), но не исключают возможности переполнения полностью, хотя и портят атакующему жизнь;
- межсетевые экраны отсекают лишь самых примитивнейших из червей, загружающих свой хвост через отдельное TCP/IP-соединение, отследить же передачу данных в контексте уже установленных TCP/IP-соединение никакой межсетевой экран не в силах.

Существуют сотни тысяч публикаций, посвященных проблеме переполнения, краткий список которых был приведен в предыдущей статье этого цикла. Среди них есть как уникальные работы, так и откровенный «поросячий визг», подогреваемый собственной крутизной (мол, смотрите, я тоже стек сорвал! Ну и что, что в лабораторных условиях?!). Статьи теоретиков от программирования элементарно распознаются замалчиванием проблем, с которыми сразу же сталкиваешься при анализе полновесных приложений и проектировании shell-кодов (по сути своей являющихся высокоавтономными роботами).

Большинство авторов ограничиваются исключительно вопросами последовательного переполнения автоматических буферов, оттесняя остальные виды переполнений на задний план, в результате чего у многих читателей создается выхолощенное представление о проблеме. На самом деле, мир переполняющихся буферов значительно шире, многограннее и интереснее, в чем мы сейчас и убедимся.

Похороненный под грудой распечаток исходного и дизассемблерного кода

Как происходит поиск переполняющихся буферов и как осуществляется проектирование shell-кода? Первым делом выбирается объект нападения, роль которого играет уязвимое приложение. Если вы хотите убедиться в собственной безопасности или атаковать строго определенный узел, вы должны исследовать конкретную версию конкретного программного пакета, установленного на конкретной машине. Если же вы стремитесь прославиться на весь мир или пытаетесь сконструировать мощное оружие, дающее вам контроль над десятками тысяч, а то и миллионами машин, ваш выбор становится уже не так однозначен.

С одной стороны, это должна быть широко распространенная и по возможности малоисследованная программа, исполняющаяся с наивысшими привилегиями и сидящая на портах, которые не так-то просто закрыть. Ра-

зумеется, с точки зрения межсетевого экрана все порты равноценны и ему абсолютно все равно, что закрывать. Однако пользователи сетевых служб и администраторы придерживаются другого мнения. Если от 135-порта, используемого червем Love San, в подавляющем большинстве случаев можно безболезненно отказаться (лично автор статьи именно так и поступил), то без услуг того же веб-сервера обойдешься едва ли.

Чем сложнее исследуемое приложение, тем больше вероятность обнаружить в нем критическую ошибку. Следует также обращать внимание и на формат представления обрабатываемых данных. Чаще всего переполняющиеся буфера обнаруживаются в синтаксических анализаторах, выполняющих парсинг текстовых строк, однако большинство этих ошибок уже давно обнаружено и устранено. Лучше искать переполняющиеся буфера там, где до вас их никто не искал. Народная мудрость утверждает: хочешь что-то хорошо спрятать – положи это на самое видное место. На фоне нашумевших эпидемий Love San и Slapper с этим трудно не согласиться. Кажется невероятным, что такие очевидные переполнения до последнего времени оставались необнаруженными!

Наличие исходных текстов одновременно желательно и нежелательно. Желательно – потому что они существенно упрощают и ускоряют поиск переполняющихся буферов, а нежелательно… по той же самой причине! Как говорится: больше народу – меньше кислороду. Действительно, трудно рассчитывать найти что-то новое в исходнике, зачитанном всеми до дыр. Отсутствие исходных текстов существенно ограничивает круг исследователей, отсекая многочисленную армию прикладных программистов и еще большую толпу откровенных непрофессионалов. Здесь, в прокуренной атмосфере ассемблерных команд, выживает лишь тот, кто программирует быстрее, чем думает, а думает быстрее, чем говорит. Тот, кто может удержать в голове сотни структур данных, буквально на физическом уровне ощущая их взаимосвязь и каким-то шестым чувством угадывая, в каком направлении нужно копать. Собственный программистский опыт может только приветствоваться. Во-первых, так легче вжиться в привычки, характер и образ мышления разработчика исследуемого приложения. Задумайтесь: а как бы вы решили данную задачу, окажись на его месте? Какие бы могли допустить ошибки? Где бы проявили непростительную небрежность, соблазнившись компактностью кода и элегантностью листинга?

Кстати об элегантности. Бытует мнение, что неряшливый стиль программного кода неизбежно провоцирует программиста на грубые ошибки (и ошибки переполнения в том числе). Напротив, педантично причесанная программа ошибок скорее всего не содержит, и анализировать ее означает напрасно тратить свое время. Как знать… Автору приходилось сталкиваться с вопиюще небрежными листингами, которые работали как часы, потому что были сконструированы настоящими профессионалами, наперед знающими, где подстелить соломку, чтобы не упасть. Встречались и по-академически аккуратные программы, дотошно и не по одному разу проверяющие все, что только можно проверить, но буквально нашпигованные ошибками переполнения. Тщательность сама по себе еще ни от чего не спасает. Для предотвращения ошибок нужен богатый программистский опыт, и опыт, оставленный граблями в том числе. Но вместе с опытом зачастую появляется и вальяжная небрежность – своеобразный «отходняк» от юношеского увлечения эффективностью и оптимизацией.

Признаком откровенного непрофессионализма является пренебрежение #define или безграмотное использование последних. В частности, если размер буфера buff определяется через MAX_BUF_SIZE, то и размер копируемой в него строки должен ограничиваться им же, а не MAX_STR_SIZE, заданным в отдельном define. Обращайте внимание и на характер аргументов функций, работающих с блоками данных.

Передача функции указателя без сообщения размера блока – частая ошибка начинающих, равно как и злоупотребление функциями strcpy/strncpy. Первая небезопасна (отсутствует возможность ограничить предельно допустимую длину копируемой строки), вторая ненадежна (отсутствует возможность оповещения о факте «обрезания» хвоста строки, не уместившегося в буфер, что само по себе может служить весьма нехилым источником ошибок).

Хорошо, ошибка переполнения найдена. Что дальше? А дальше только дизассемблер. Не пытайтесь выжать из исходных текстов хоть какую-то дополнительную информацию. Порядок размещения переменных в памяти не определен и практически никогда не совпадает с порядком их объявления в программе. Может оказаться так, что большинства из этих переменных в памяти попросту нет и они размещены компилятором в регистрах либо же вовсе отброшены оптимизатором как ненужные. (Попутно заметим, что все демонстрационные листинги, приведенные в этой статье, рассчитывают, что переменные располагаются в памяти в порядке их объявления.)

Впрочем, не будем забегать вперед – дизассемблирование – тема отдельной статьи, которую планируется опубликовать в следующих номерах журнала.

Цели и возможности атаки

Конечная цель любой атаки – заставить систему сделать что-то «нехорошее», чего нельзя добиться легальным путем. Существует по меньшей мере четыре различных способа реализации атаки:

- чтение секретных переменных;
- модификация секретных переменных;
- передача управления на секретную функцию программы;
- передача управления на код, переданный жертве самим злоумышленником.

Чтение секретных переменных. На роль секретных переменных в первую очередь претендуют пароли на вход в систему, а также пароли доступа к конфиденциальной информации. Все они так или иначе содержатся в адресном пространстве уязвимого процесса, зачастую располагаясь по фиксированным адресам (под «входом в систему» здесь подразумеваются имя пользователя и

пароль, обеспечивающие удаленное управление уязвимым приложением).

Еще в адресном пространстве процесса содержатся дескрипторы секретных файлов, сокеты, идентификаторы TCP/IP-соединений и многое другое. Разумеется, вне текущего контекста они не имеют никакого смысла, но могут быть использованы кодом, переданном жертве злоумышленником, и осуществляющим, например, установку «невидимого» TCP/IP-соединения, прячась под «крышей» уже существующего.

Ячейки памяти, хранящие указатели на другие ячейки, «секретными» строго говоря не являются, однако, знание их содержимого значительно облегчает атаку. В противном случае атакующему придется определять опорные адреса вслепую. Допустим, в уязвимой программе содержится следующий код:

```
char *p = malloc(MAX_BUF_SIZE);
```

где p – указатель на буфер, содержащий секретный пароль. Допустим так же, что в программе имеется ошибка переполнения, позволяющая злоумышленнику читать содержимое любой ячейки адресного пространства. Весь вопрос в том: как этот буфер найти? Сканировать всю кучу целиком не только долго, но и небезопасно, т.к. можно легко натолкнуться на невыделенную страницу памяти и тогда выполнение процесса аварийно завершится. Автоматические и статические переменные в этом отношении более предсказуемы. Поэтому атакующий должен сначала прочитать содержимое указателя p, а уже затем – секретный пароль, на который он указывает. Разумеется, это всего лишь пример, которым возможности переполняющего чтения не ограничиваются.

Само же переполняющее чтение реализуется по меньшей мере четырьмя следующими механизмами: «потерей» завершающего нуля в строковых буферах, модификаций указателей (см. «Указатели и индексы»), индексным переполнением (см. там же) и навязыванием функции printf (и другим функциям форматированного вывода) лишних спецификаторов.

Модификация секретных переменных. Возможность модификации переменных дает значительно больше возможностей для атаки, позволяя:

- навязывать уязвимой программе «свои» пароли, дескрипторы файлов, TCP/IP-идентификаторы и т. д.;
- модифицировать переменные, управляющие ветвлением программы;
- манипулировать индексами и указателями, передавая управление по произвольному адресу (и адресу, содержащему код, специально подготовленный злоумышленником в том числе).

Чаще всего модификация секретных переменных реализуется посредством последовательного переполнения буфера, по обыкновению своему поражающего целый каскад побочных эффектов. Например, если за концом переполняющегося буфера расположен указатель на некоторую переменную, в которую после переполнения что-то пишется, злоумышленник сможет затереть любую ячейку памяти на свой выбор (за исключением ячеек, явно защищенных от модификации, например кодовой секции или секции .rodata, разумеется).

Передача управления на секретную функцию программы. Модификация указателей на исполняемый код приводит к возможности передачи управления на любую функцию уязвимой программы (правда, с передачей аргументов имеются определенные проблемы). Практически каждая программа содержит функции, доступные только root, и предоставляющие те или иные управленческие возможности (например, создание новой учетной записи, открытие сессии удаленного управления, запуск файлов и т. д.). В более изощренных случаях управление передается на середину функции (или даже на середину машинной инструкции) с таким расчетом, чтобы процессор выполнил замысел злоумышленника, даже если разработчик программы не предусматривал ничего подобного.

Передача управления обеспечивается либо за счет изменения логики выполнения программы, либо за счет подмены указателей на код. И то, и другое опирается на модификацию ячеек программы, кратко рассмотренную выше.

Передача управления на код, переданный жертве самим злоумышленником, является разновидностью механизма передачи управления на секретную функцию программы, только сейчас роль этой функции выполняет код, подготовленный злоумышленником и тем или иным способом переданный на удаленный компьютер. Для этой цели может использоваться как сам переполняющийся буфер, так и любой другой буфер, доступный злоумышленнику для непосредственной модификации и в момент передачи управления на shell-код присутствующий в адресном пространстве уязвимого приложения (при этом он должен располагаться по более или менее предсказуемым адресам, иначе передавать управление будет некому и некуда).

Жертвы переполнения или объекты атаки

Переполнение может затирать ячейки памяти следующих типов: указатели, скалярные переменные и буфера. Объекты языка Си++ включают в себя как указатели (указывающие на таблицу виртуальных функций, если таковые в объекте есть), так и скалярные данные-члены (если они есть). Самостоятельной сущности они не образуют и вполне укладываются в приведенную выше классификацию.

Указатели и индексы

В классическом Паскале и других «правильных» языках указатели отсутствуют, но в Си/Си++ они вездесущи. Чаще всего приходится иметь дело с указателями на данные, несколько реже встречаются указатели на исполняемый код (указатели на виртуальные функции, указатели на функции, загружаемые динамической компоновкой и т. д.). Современный Паскаль (раньше ассоциируемый с компилятором Turbo Pascal, а теперь еще и DELPHI) также немыслим без указателей. Даже если в явном виде указатели и не поддерживаются, на них дер-

жатся динамические структуры данных (куча, разряженные массивы), используемые внутри языка.

Указатели удобны. Они делают программирование простым, наглядным, эффективным и естественным. В то же время указатели во всех отношениях категорически небезопасны. Попав в руки хакера или пищеварительный тракт червя, они превращаются в оружие опустошительной мощности – своеобразный аналог BFG-900 или, по крайней мере, плазмогана. Забегая вперед, отметим, что указатели обоих типов потенциально способны к передаче управления на несанкционированный машинный код.

Вот с указателей на исполняемый код мы и начнем. Рассмотрим ситуацию, когда следом за переполняющимся буфером buff, расположен указатель на функцию, которая инициализируется до и вызывается после переполнения буфера (возможно, вызывается не сразу, а спустя некоторое время). Тогда мы заполучим аналог функции call или, говоря другими словами, инструмент для передачи управления по любому (ну или почти любому) машинному адресу, в том числе и на сам переполняющийся буфер (тогда управление получит код, переданный злоумышленником).

Листинг 3. Фрагмент программы, подверженной переполнению с затиранием указателя на исполняемый код

```
code_ptr()
{
    char buff[8]; void (*some_func) ();
    …
    printf("passws:"); gets(buff);
    …
    some_func();
}
```

Подробнее о выборе целевых адресов мы поговорим в другой раз, сейчас же сосредоточимся на поиске затираемых указателей. Первым в голову приходит адрес возврата из функции, находящийся внизу кадра стека (правда, чтобы до него дотянуться, требуется пересечь весь кадр целиком и не факт, что нам это удастся, к тому же его целостность контролируют многие защитные системы, подробнее о которых планируется рассказать в следующей статье).

Другая популярная мишень – указатели на объекты. В Си++ программах обычно присутствует большое количество объектов, многие из которых создаются вызовом оператора new, возвращающим указатель на свежесозданный экземпляр объекта. Невиртуальные функции-члены класса вызываются точно так же, как и обычные Си-функции (т.е. по их фактическому смещению), поэтому они неподвластны атаке. Виртуальные функции-члены вызываются намного более сложным образом через цепочку следующих операций: указатель на экземпляр объекта → указатель на таблицу виртуальных функций → указатель на конкретную виртуальную функцию. Указатели на таблицу виртуальных функций не принадлежат объекту и внедряются в каждый его экземпляр, который чаще всего сохраняется в оперативной памяти, реже – в регистровых переменных. Указатели на объекты так же размещаются либо в оперативной памяти, либо в регистрах, при этом на один и тот же объект может указывать множество указателей (среди которых могут встретиться и такие, которые расположены непосредственно за концом переполняющегося буфера). Таблица виртуальных функций (далее просто виртуальная таблица) принадлежит не экземпляру объекта, а самому объекту, т.е. упрощенно говоря, мы имеем одну виртуальную таблицу на каждый объект. «Упрощенно» потому, что в действительности виртуальная таблица помещается в каждый obj-файл, в котором встречается обращение к членам данного объекта (раздельная компиляция дает о себе знать). И хотя линкеры в подавляющем большинстве случаев успешно отсеивают лишние виртуальные таблицы, иногда они все-таки дублируются (но это уже слишком высокие материи для начинающих). В зависимости от «характера» выбранной среды разработки и профессионализма программиста виртуальные таблицы размещаются либо в секции .data (не защищенной от записи), либо в секции .rodata (доступной лишь на чтение), причем последний случай встречается значительно чаще.

Давайте для простоты рассмотрим приложения с виртуальными таблицами в секции .data. Если злоумышленнику удастся модифицировать один из элементов виртуальной таблицы, то при вызове соответствующей виртуальной функции управление получит не она, а совсем другой код! Однако добиться этого будет непросто! Виртуальные таблицы обычно размещаются в самом начале секции данных, т.е. перед статическими буферами и достаточно далеко от автоматических буферов (более конкретное расположение указать невозможно, т.к. в зависимости от операционной системы стек может находиться как ниже, так и выше секции данных). Так что последовательное переполнение здесь непригодно и приходится уповать на индексное, все еще остающееся теоретической экзотикой, робко познающей окружающий мир.

Модифицировать указатель на объект и/или указатель на виртуальную таблицу намного проще, поскольку они не только находятся в области памяти, доступной для модификации, но и зачастую располагаются в непосредственной близости от переполняющихся буферов.

Модификация указателя this приводит к подмене виртуальных функций объекта. Достаточно лишь найти в памяти указатель на интересующую нас функцию (или вручную сформировать его в переполняющемся буфере) и установить на него this с таким расчетом, чтобы адрес следующей вызываемой виртуальной функции попал на подложный указатель. С инженерной точки зрения это достаточно сложная операция, поскольку кроме виртуальных функций объекты еще содержат и переменные, которые более или менее активно используют. Переустановка указателя this искажает их «содержимое» и очень может быть, что уязвимая программа рухнет раньше, чем успеет вызвать подложную виртуальную функцию. Можно, конечно, сымитировать весь объект целиком, но не факт, что это удастся. Сказанное относится и к указателю на объект, поскольку с точки зрения компилятора они скорее похожи, чем различны. Однако наличие двух различных сущностей дает атакующему свободу выбора – в некоторых случаях предпочтительнее затирать указатель this, в некоторых случаях – указатель на объект.

Листинг 4. Фрагмент программы, подверженной последовательному переполнению при записи, с затиранием указателя на таблицу виртуальных функций

```
class A{
public:
    virtual void f() { printf("legal\n");};
};

main()
{
    char buff[8]; A *a = new A;
    printf("passwd:");gets(buff); a->f();
}
```

Листинг 5. Дизассемблерный листинг переполняющийся программы с краткими комментариями

```
; CODE XREF: start+AF↓p
.text:00401000 main      proc near
.text:00401000
.text:00401000 var_14    = dword ptr -14h      ; this
.text:00401000 var_10    = dword ptr -10h      ; *a
.text:00401000 var_C     = byte ptr -0Ch
.text:00401000 var_4     = dword ptr -4
.text:00401000
.text:00401000                   push    ebp
.text:00401001                   mov     ebp, esp
.text:00401003                   sub     esp, 14h
; открываем кадр стека и резервируем 14h стековой памяти
.text:00401003
.text:00401003 ;
.text:00401006                   push    4
.text:00401008                   call    operator new(uint)
.text:0040100D                   add     esp, 4
; выделяем память для нового экземпляра объекта A и получаем
; указатель
.text:0040100D
.text:0040100D ;
.text:00401010                   mov     [ebp+var_10], eax
; записываем указатель на объект в переменную var_10
.text:00401010
.text:00401010 ;
.text:00401013                   cmp     [ebp+var_10], 0
.text:00401017                   jz      short loc_401026
.text:00401017 ; проверка успешности выделения памяти
.text:00401017 ;
.text:00401019                   mov     ecx, [ebp+var_10]
.text:0040101C                   call    A::A
.text:0040101C ; вызываем конструктор объекта A
.text:0040101C ;
.text:00401021                   mov     [ebp+var_14], eax
; заносим возвращенный указатель this в переменную var_14
.text:00401021
.text:00401021 ;
…
.text:0040102D loc_40102D:       ; CODE XREF: main+24↑j
.text:0040102D                   mov     eax, [ebp+var_14]
.text:00401030                   mov     [ebp+var_4], eax
; берем указатель this и перепрятываем его в переменную var_4
.text:00401030
.text:00401030 ;
; "passwd:"
.text:00401033                   push    offset aPasswd
.text:00401038                   call    _printf
.text:0040103D                   add     esp, 4
.text:0040103D ; выводим приглашение к вводу на экран
.text:0040103D ;
.text:00401040                   lea     ecx, [ebp+var_C]
; переполняющийся буфер расположен ниже указателя на объект
; и первичного указателя this, но выше порожденного указателя
; this, что делает последний уязвимым
.text:00401040
.text:00401040
.text:00401040
.text:00401040    ;
.text:00401043                   push    ecx
.text:00401044                   call    _gets
.text:00401049                   add     esp, 4
.text:00401049 ; чтение строки в буфер
.text:00401049 ;
.text:0040104C                   mov     edx, [ebp+var_4]
; загружаем уязвимый указатель this в регистр EDX
.text:0040104C
.text:0040104C ;
.text:0040104F                   mov     eax, [edx]
.text:0040104F ; извлекаем адрес виртуальной таблицы
.text:0040104F ;
.text:00401051                   mov     ecx, [ebp+var_4]
.text:00401051 ; передаем функции указатель this
.text:00401051 ;
.text:00401054                   call    dword ptr [eax]
; вызываем виртуальную функцию – первую функцию виртуальной
; таблицы
.text:00401054
.text:00401054 ;
.text:00401056                   mov     esp, ebp
.text:00401058                   pop     ebp
.text:00401059                   retn
.text:00401059 main              endp
```

Рассмотрим ситуацию, когда следом за переполняющимся буфером идет указатель на скалярную переменную p и сама переменная x, которая в некоторый момент выполнения программы по данному указателю и записывается (порядок чередования двух последних переменных несущественен, главное, чтобы переполняющийся буфер затирал их всех). Допустим также, что с момента переполнения ни указатель, ни переменная не претерпевают никаких изменений (или изменяются предсказуемым образом). Тогда, в зависимости от состояния ячеек, затирающих оригинальное содержимое переменных x и p, мы сможем записать любое значение x по произвольному адресу p, осуществляя это «руками» уязвимой программы. Другими словами, мы получаем аналог функций POKE и PatchByte/PatchWord языков Бейсик и IDA-Си соответственно. Вообще-то, на выбор аргументов могут быть наложены некоторые ограничения (например, функция gets не допускает символа нуля в середине строки), но это не слишком жесткое условие и имеющихся возможностей вполне достаточно для захвата управления над атакуемой системой.

Листинг 6. Фрагмент программы, подверженной последовательному переполнению при записи и затиранием скалярной переменной и указателя на данные, поглощающими затертую переменную

```
data_ptr()
{
    char buff[8]; int x; int *p;
    printf("passws:"); gets(buff);
    …
    *p = x;
}
```

Индексы являются своеобразной разновидностью указателей. Грубо говоря, это относительные указатели, адресуемые относительно некоторой базы. Смотрите, p[i] можно представить и как *(p+i), практически полностью уравнивая p и i в правах.

Модификация индексов имеет свои слабые и сильные стороны. Сильные – указатели требуют задания абсолютного адреса целевой ячейки, который обычно неизвестен, в то время как относительный вычисляется на ура. Индексы, хранящиеся в переменных типа char, лишены проблемы нулевых символов. Индексы, хранящиеся в переменных типа int, могут беспрепятственно затирать ячейки, расположенные «выше» стартового адреса (т.е. лежащие в младших адресах), при этом старшие байты индекса содержат символы FFh, которые значительно более миролюбивы, чем символы нуля.

Однако, если обнаружить факт искажения указателей практически невозможно (дублировать их значение в резервных переменных не предлагать), то оценить корректность индексов перед их использованием не составляет

никакого труда, и многие программисты именно так и поступают (правда, «многие» еще не означает «все»). Другой слабой стороной индексов является их ограниченная «дальнобойность», составляющая ±128/256 байт (для индексов типа signed/unsigned char) и -2147483648 байт для индексов типа signed int.

```
Листинг 7. Фрагмент программы, подверженной последовательному
переполнению при записи, с затиранием индекса

index_ptr()
{
    char *p; char buff[MAX_BUF_SIZE]; int i;
    p = malloc(MAX_BUF_SIZE); i = MAX_BUF_SIZE;
    …
    printf("passws:"); gets(buff);
    …
    // if ((i < 1) || (i > MAX_BUF_SIZE)) ошибка
    while(i--) p[i] = buff[MAX_BUF_SIZE - i];
}
```

Скалярные переменные

Скалярные переменные, не являющиеся ни индексами, ни указателями, намного менее интересны для атакующих, поскольку в подавляющем большинстве случаев их возможности очень даже ограничены, однако на безрыбье сгодятся и они (совместное использование скалярных переменных вместе с указателями/индексами мы только что рассмотрели, сейчас же нас интересуют скалярные переменные сами по себе).

Рассмотрим случай, когда вслед за переполняющимся буфером расположена переменная buks, инициализируемая до переполнения, а после переполнения используемая для расчетов количества денег, снимаемых со счета (не обязательно счета злоумышленника). Допустим, программа тщательно проверяет входные данные и не допускает использования ввода отрицательных значений, однако, не контролирует целостность самой переменной buks. Тогда, варьируя ее содержимым по своему усмотрению, злоумышленник без труда обойдет все проверки и ограничения.

```
Листинг 8. Фрагмент программы, подверженный переполнению
с затиранием скалярной переменной

var_demo(float *money_account)
{
    char buff[MAX_BUF_SIZE]; float buks = CURRENT_BUKS_RATE;
    printf("input money:"); gets(buff);
    if (atof(buff)<0) ошибка! введите положительное значение
    …
    *money_account -= (atof(buff) * CURRENT_BUKS_RATE);
}
```

При всей своей искусственности приведенный пример чрезвычайно нагляден. Модификация скалярных переменных только в исключительных случаях приводит к захвату управления системой, но легко позволяет делать из чисел винегрет, а на этом уже можно сыграть! Но что же это за исключительные случаи? Во-первых, многие программы содержат отладочные переменные, оставленные разработчиками, и позволяющие, например, отключить систему аутентификации. Во-вторых, существует множество переменных, хранящих начальные или предельно допустимые значения других переменных, например, счетчиков цикла – for (a =b; a < c; a++) *p++ = *x++; очевидно, что модификация переменных b и c приведет к переполнению буфера p со всеми отсюда вытекающими последствиями. В-третьих… да мало ли что можно придумать – всего и не перечислишь! Затирание скалярных переменных при переполнении обычно не приводит к немедленному обрушению программы, поэтому такие ошибки могут долго оставаться не обнаруженными. Будьте внимательными!

Массивы и буфера

Что интересного можно обнаружить в буферах? Прежде всего это строки, хранящиеся в PASCAL-формате, т.е. с полем длины вначале, затирание которого порождает каскад вторичных переполнений. Про уязвимость буферов с конфиденциальной информацией мы уже говорили, а теперь, пожалуйста – конкретный, хотя и несколько наигранный пример:

```
Листинг 9. Фрагмент программы, подверженной последовательному
переполнению при записи с затиранием постороннего буфера

buff_demo()
{
    char buff[MAX_BUF_SIZE];
    char pswd[MAX_BUF_SIZE];
    …
    fgets(pswd, MAX_BUF_SIZE, f);
    …
    printf("passwd:"); gets(buff);
    if (strncmp(buff, pwsd, MAX_BUF_SIZE))
                        // неправильный пароль
    else
                        // правильный пароль
}
```

Еще интересны буфера, содержащие имена открываемых файлов (можно заставить приложение записать конфиденциальные данные в общедоступный файл или, напротив, навязать общедоступный файл взамен конфиденциального), тем более что несколько подряд идущих буферов, вообще говоря, не редкость.

Заключение

Изначально статья задумывалась как исчерпывающее руководство, снабженное большим количеством листингов и избегающее углубляться в академические теоретизирования. Теперь, вычитывая статью перед заключительной правкой и внося в нее мелкие, косметические улучшения, я с грустью осознаю, что выполнить свой замысел мне так и не удалось… До практических советов разговор вообще не дошел, и львиная часть подготовленного материала осталась за кадром. Ох, и не следовало мне пытаться объять необъятное…

Надеюсь, что следующие статьи этого цикла исправят положение. В первую очередь планируется рассказать о передовых методиках переполнения кучи, приводящих к захвату управления удаленной машиной, обсудить технические аспекты разработки shell-кода (приемы создания позиционно-независимого кода, проблема вызова системных функций, оптимизация размера и т. д.), затем можно будет перейти к разговору о способах поиска переполняющихся буферов и о возможных мерах по заблаговременному предотвращению оных. Отдельную статью хотелось бы посвятить целиком червям Love Sun и Slapper, поскольку их дизассемблерные листинги содержат очень много интересного.