Денис Силаков

Ошибки сегментации и безопасность, или как упавшая программа может привести к взлому системы

Многие пользователи ОС Linux (и не только ее) наверняка хотя бы раз в жизни сталкивались с “падением” того или иного приложения. Сам факт аварийного завершения программы уже неприятен – ведь он может привести к потере важных данных, вызвать необходимость заново повторять уже проделанную работу и так далее. Однако последствия могут быть еще более печальны – наличие программ, в некоторых ситуациях завершающихся аварийно, может быть серьезной угрозой безопасности всей системы. То есть “падающая” программа – это с достаточно большой вероятностью брешь в безопасности ОС. Почему так? Давайте разберемся.

Модель памяти процесса и атаки на переполнение буфера

Одной из основных причин аварийного завершения работы программ вот уже несколько десятилетий является некорректная работа с памятью, приводящая, как правило, к ошибке сегментации (segmentation fault) — попытке обращения к несуществующему адресу. Для понимания причин возникновения таких проблем и исходящих от них угроз для безопасности системы, необходимо рассмотреть организацию памяти любого процесса в Linux на основных аппаратных архитектурах.

Память любой программы представляется как линейный массив байтов. В начало этого массива помещается код программы, за ним следуют данные – глобальные переменные (как инициализированные, так и не имеющие начального значения) и так называемая “куча” («Heap»), из которой в процессе работы программы выделяется память под динамические переменные. Куча “растет” в сторону увеличения адресов. Ближе к концу массива располагается стек, который растет в сторону уменьшения адресов (навстречу куче).

Посмотреть расположение кучи, стека и ряда других объектов (например, загруженных разделяемых библиотек) в памяти процесса можно посмотреть с помощью файловой системы procfs – вся эта информация содержится в файле /proc/<PID>/maps, где <PID> - идентификатор процесса.

Стек

В стек записываются локальные переменные функций и вспомогательная информация, используемая при их вызове – в частности, адрес возврата и аргументы. Информация, относящаяся к вызову одной функции, формирует *фрейм*. Чтобы определить границы фреймов, в каждый новый фрейм помещается указатель на предыдущий. Структуру стека при вызове некоторой функции можно представить диаграммой, представленной на Рисунке 1.

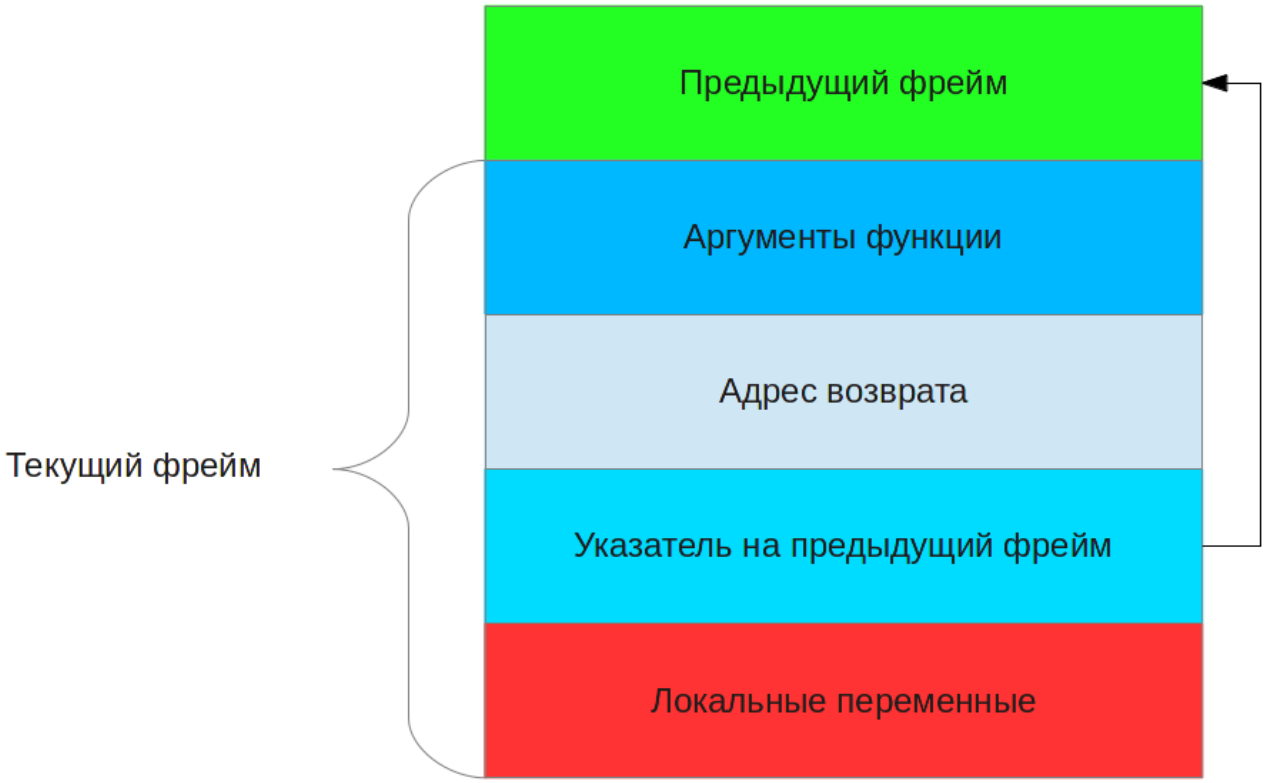

Рисунок 1. Структура стека при вызове функции.

Адреса на этой диаграмме увеличиваются снизу вверх, а сам стек растет «сверху вниз» (то есть адрес предыдущего фрейма больше адреса текущего). Если в области локальных переменных есть переменная-буфер, в которую последовательно записываются некоторое данные, то буфер будет «расти» снизу вверх. И если записать в в этот буфер слишком много информации, то она может затереть значения, располагающиеся на более высоких адресах.

В частности, могут оказаться перезаписанными указатель на предыдущий фрейм и значение адреса, по которому необходимо производить возврат из функции. В результате вместо реального адреса система получит некоторое значение, с большой вероятностью не имеющее особого смысла. Попытка передать управление на этот адрес скорее всего приведет либо к ошибке сегментации, либо к ошибке некорректной инструкции (Illegal instruction).

Однако если окажется, что по новому “адресу” находится некоторый код, который может быть исполнен, то работа приложения продолжится – правда, совсем не так, как ожидали пользователи или разработчики. Именно на этом и основаны многие атаки, суть которых заключается в использовании переполнения буфера данных для подмены адреса возврата из функции. В случае успешной атаки, старый адрес подменяется новым, по которому располагается некоторая вредоносная функция. В принципе, после выполнения своих вредоносных действий (например, отсылки ваших персональных данных куда-нибудь “на сторону”) эта функция может передать выполнение на настоящий адрес – так что приложение даже имеет шанс продолжить работать (конечно, если в результате переполнения не возникло других повреждений данных в памяти, критичных для работы программы).

Порядок расположения переменных на стеке в момент вызова функции вычислить нетрудно (он определяется соглашениями вызова функций, используемыми в компиляторах GCC). Также

нетрудно просчитать - что, как и куда надо записать, чтобы переполнить буфер «с умом», перезаписав нужные переменные нужными значениями.

Куча

Схожим образом можно организовать и атаки на переполнение буферов, располагающихся в куче. Ведь один из часто используемых типов данных — это указатель на функцию, и некорректное изменение какого-либо указателя может привести к краху программы или вызову “неожиданной” функции.

С перезаписью нужных переменных могут возникнуть сложности, поскольку при распределении памяти в куче могут применяться достаточно сложные алгоритмы, и гарантированно вычислить взаимное расположение переменных в памяти можно не всегда. Поэтому далеко не всякую ошибку сегментации в куче легко обратить в цель для атаки. Впрочем, если атакуемый буфер располагается «недалеко» от переменной, которую нужно перезаписать (например, внутри одной структуры или класса), то проблем возникнуть не должно.

Например, рассмотрим следующий код:

```
#include <stdio.h>
void f() {
  printf("Here I am!\n");
}

struct A {
  char str[2];
  void(*func)(void);
} a;

void main() {
    a.func = f;
    gets(a.str);
    a.func();
}
```

Здесь мы считываем с помощью функции gets() в поле a.str строку, никак не контролируя ее длину, в то время как размер поля составляет всего два байта. В процессе выполнения считываемая строка будет записываться в структуру, начиная с первого байта поля str. И начиная с некоторого размера строки, она «залезет» на поле func (в зависимости от разрядности вашей системы и опций компиляции, влияющих на выравнивание полей структур в памяти, критическая длина строки может оказаться различной). В результате, при вызове a.func() процессор попробует выполнить отнюдь не функцию f(). Нетрудно и подобрать такую строку, которая перезатрет поле a.func чем-то осмысленным. Например, в моей рабочей системе мне удалось подобрать строку, ввод которой приводил к повторному вызову gets().

В заключение отмечу, что помимо стека и динамических переменных в куче, популярной целью для атак являются таблицы GOT (Global Offset Table) и PLT (Procedure Linkage Table). Такие таблицы есть у каждого процесса, использующего динамически слинкованные библиотеки (а практически все бинарные приложения в дистрибутивах Linux используют те или иные библиотеки). Таблицы GOT и PLT указывают, по каким адресам в памяти процесса находятся

библиотечные данные и функции. Они располагаются в области данных процесса и могут быть изменены злоумышленником, в результате чего адрес некоторой библиотечной функции будет подменен адресом вредоносной процедуры.

Средства защиты и способы их обхода

Переполнение буфера — очень распространенная проблема, встречающаяся во множестве приложений, созданных самыми разными разработчиками. Нельзя уповать на то, что все программисты вдруг перестанут делать такие ошибки. Также пока не предвидится появления надежных средств обнаружения проблемных мест в приложениях. Возможным решением является использование языков с автоматическим управлением памятью (типа Java, Perl/Python, семейства .NET и им подобных), где за соблюдением границ всех переменных следит соответствующий интерпретатор. Но разработчики не спешат забрасывать C/C++ и полностью переходить на подобные технологии — на то есть веские причины, но их обсуждение выходит за рамки данной статьи.

В общем, глобального избавления от первоисточника проблем и устранения всех ошибок в приложениях не предвидится. Но правка кода проблемных приложений — не единственный способ обезопасить ОС и другие программы. За десятилетия развития ИТ был предложен не один подход к решению этой проблемы на общесистемном уровне. Рассмотрим наиболее распространенные из них.

Слова-канарейки

Одним из популярных и доступных способов предотвратить эксплуатацию уязвимостей, связанных с переполнением буфера в стеке при вызове функций, является использование “слов-канареек” (“canary words”). Способ заключается в записи некоторого специального значения в ячейку памяти, следующей за выделенным под данные буфером, который может быть модифицирован в ходе работы функции. Перед возвратом из функции, система осуществляет проверку целостности этого значения. В случае, если буфер переполнился, “канарейка” затрется каким-то другим значением (конечно, есть шанс, что новое значение совпадет с предыдущим, но вероятность этого крайне мала).

Нарушение целостности слова-канарейки просигнализирует системе, что адрес возврата из функции (либо другие данные, следующие за переполнившимся буфером) может быть переписан. При обнаружении такого нарушения перехода по адресу возврата не производится; вместо этого программа аварийно завершается.

Свое название способ получил из-за схожести с настоящими канарейками, которых шахтеры брали в угольные выработки для обнаружения выходов угарного газа — в подобных ситуациях канарейки умирали первыми, что служило сигналом о необходимости срочно покинуть опасное место.

В Linux использовать подобную защиту достаточно просто — для этого в компиляторах GCC предусмотрены опции -fstack-protector и -fstack-protector-all, создающие «канареек» для всех потенциально опасных функций приложения. Если приложение собрано с такими опциями, то при обнаружении переполнения буфера оно будет остановлено с ошибкой вида:

```
*** stack smashing detected ***:  program terminated
```

Однако переполнение может возникать при работе функций из сторонних библиотек, используемых в программе. В таком случае о «канарейках» должны позаботиться создатели

библиотеки, а точнее — люди, занимающиеся ее сборкой для конкретной системы. В дистрибутивах, распространяющихся в виде набора прекомпилированных пакетов, это могут сделать сами разработчики, а в source-based системах все зависит от флагов компиляции, выставленных пользователем.

Поскольку слова-канарейки имеют и очевидный минус в виде уменьшения производительности, то далеко не всегда они используются по умолчанию — ведь поголовное их использование может сказаться на таких важных для пользователя факторах, как скорость загрузки системы или время выхода из спящего режима. Поэтому не стоит рассчитывать на то, что все библиотеки в системе собраны с поддержкой canary words.

«Канарейки» в glibc

В основных системных библиотеках Linux имеется немного иная схема реализации подобных средств защиты. А именно, в библиотеках glibc для большинства функций, вызов которых может привести к переполнению буфера, предлагаются альтернативные реализации, использующие слова-канарейки для проверки целостности стека. Такие альтернативы имеют суффикс '_chk' - printf_chk(), scanf_chk() и так далее. У программистов нет нужды образаться к этим функциям напрямую; их вызов подставляется компилятором в код программы автоматически вместо обычных функций при выставлении константы _FORTIFY_SOURCE и только если включена оптимизация. При срабатывании этих «канареек», мы получим схожее сообщение о переполнении буфера («*** buffer overflow detected ***»), а заодно и трассу вызова функций и карту памяти процесса (фактически - содержимое уже упоминавшегося файла /proc/<PID>/maps).

Более того, при выставлении константы _FORTIFY_SOURCE некоторые ошибки можно отловить еще на этапе компиляции — если компилятор определит, что переполнение будет возникать в любом случае (например, вы пытаетесь записать строку из трех символов в массив из двух байтов), он выдаст соответствующее предупреждение.

Многие современные дистрибутивы общего назначения выставляют _FORTIFY_SOURCE при сборке программ в своих репозиториях. На производительности системных компонентов это не сказывается, для большинства приложений замедление также практически незаметно, зато безопасность системы повышается достаточно серьезно.

В целом, слова-канарейки — надежное средство защиты, которое достаточно сложно обойти и которое позволяет достаточно эффективно обнаруживать переполнения. Но далеко не для каждого потенциально опасного буфера можно «посадить» канареек средствами компилятора и системных библиотек. К тому же использование канареек сопряжено с дополнительными накладными расходами, поэтому в приложениях, где критична производительность, их применение может оказаться нежелательным.

Однако есть еще одно средство, защищающее систему от атак на переполнение буфера на аппаратном уровне, без какого-либо участия разработчиков потенциально уязвимых приложений.

NX-бит

… И способ этот заключается в использовании NX-бита (No eXecutable bit) для страниц памяти процесса, не содержащих код. Этот подход не предполагает непосредственного обнаружения факта переполнения; вместо этого, он пытается затруднить использование подобных ошибок для исполнения вредоносного кода.

Напомню, что атака, использующая переполнение буфера, основывается на модификации потока управления программы — вместо инструкций, предусмотренных программистами, происходит передача управления на вредоносный код. Однако передать управление при этом можно только коду, находящемуся в адресном пространстве этого же процесса. Поэтому злоумышленнику требуется каким-то образом внедрить свой код непосредственно в атакуемый процесс.

На заре развития ИТ можно было бы просто перезаписать инструкции прямо в коде программы. Но как быстро показала практика, возможность перезаписывать код программы в процессе ее исполнения может принести немало печальных сюрпризов даже без участия злоумышленников. Поэтому во всех современных системах области памяти, содержащие исполняемый код, помечаются как «read-only» («только для чтения») на аппаратном уровне сразу после того, как код загружается в память. В ходе работы программы модификация этих областей памяти невозможна.

Вторым по простоте и доступности способом внедрения кода является его помещение в область данных процесса. Например, можно поместить необходимый набор байтов в какую-нибудь переменную окружения, передать вместе со считываемыми процессом входными данными и так далее. Именно этот подход и использовался долгое время в большинстве атак на переполнение буфера.

И именно от подобных внедрений и защищает NX-бит, работающий на уровне страниц памяти. (Здесь уместно напомнить, что при управлении оперативной памятью ОС и аппаратура оперируют не отдельными байтами, а *страницами* — областями достаточно большого размера, каждая из которых имеет набор вспомогательных атрибутов, в том числе и NX-бит).

Если заведомо известно, что некоторая страница памяти содержит данные (а операционной системе, загружающей программу в память, заведомо известно - где данные, а где - код), то она помечается NX-битом. Попытка исполнения кода с таких страниц блокируется на аппаратном уровне — так что старые проверенные способы внедрения кода не работают.

Естественно, NX-бит должен поддерживаться операционной системой — ведь аппаратура сама по себе не знает, где располагается код, а где данные. Во всех современных ОС общего назначения такая поддержка присутствует.

Реализация NX-бита в аппаратуре и добавление его поддержки в операционные системы стало серьезным шагом в укреплении безопасности и наверняка позволила пресечь множество атак. Однако как это нередко бывает, потребовалось совсем немного времени для создания элегантного способа обхода этой защиты. В чем же этот способ состоит?

Основной целью NX-бита является предотвращение запуска каких-либо инструкций из области данных, и с этой задачей он блестяще справляется. Однако так ли необходима возможность запуска кода из области данных для успешной атаки? Оказывается, что совсем нет. Ведь подавляющее большинство приложений в Linux динамически слинкованы с библиотекой libc, в которой есть такие замечательные функции, как system() и семейство exec(), позволяющие выполнить любую команду оболочки или запустить произвольную программу.

Так что вообще-то нет нужды создавать свои собственные функции, помещать их в область данных, а потом передавать им управление – достаточно передать управление одной из упомянутых функций libc, передав ей нужные параметры – например, вызвав system(“/bin/bash”). Этому NX-бит не мешает – ведь функция system() находится в области кода, и ее запуск легитимен. Атака, обходящая NX-бит посредством вызова функций библиотеки libc, получила название “Return-Into-LibC”.

Конечно, остаются задачи определения адреса нужной функции в области памяти процесса и формирования нужных аргументов, но они не является неразрешимой. Хотя существуют специальные приемы, дополнительно усложняющие эти задачи — такие, как технология рандомизации адресного пространства процессы (Address Space Layout Randomization), при использовании которой библиотеки при каждом старте программы располагаются в ее адресном пространстве случайным образом. Из-за этого нельзя заранее узнать адрес той или иной библиотечной функции, и атакующему приходится определять адрес нужной функции непосредственно в ходе работы программы. Впрочем, это хоть и сложно, но выполнимо (к тому же для успешной атаки и не обязательно использовать каждое переполнение буфера, достаточно одного удачного «попадания»). Так что NX-бит работает, но при большом желании обходится.

Итак, для предотвращения атак на переполнение буфера есть достаточно мощные универсальные средства, защищающие от атак все приложения системы. Однако несмотря на высокую эффективность, стопроцентную гарантию защиты они дать не могут. Существующие технологии затрудняют атаки, но отнюдь не делают их невозможными; подобные средства могут остановить «юных хакеров», но вряд ли станут серьезным препятствием для профессионалов.

Однако помимо технических средств, в мире открытого ПО есть ряд нетехнических факторов, косвенно повышающих безопасность систем конечных пользователей.

Специфика FLOSS

Нетрудно видеть, что реализация атаки на переполнение буфера сильно привязана к коду атакуемой программы – даже незначительная модификация кода (пусть и не устраняющая ошибку, но, например, сдвигающая данные в памяти, переставляющая аргументы и тому подобное) может потребовать изменения и в коде вредоносной программы. Поэтому с этой точки зрения, политика частых релизов, которой придерживаются многие приложения в мире FLOSS, играет на стороне безопасности – написать вредоносное ПО, способное атаковать все версии программы, сложнее, чем целиться на какую-то одну из них.

Добавьте к этому многообразие дистрибутивов Linux, в которых программы собираются с различными патчами, настройками среды и флагами компиляции – и получите богатую мозаику, в которой сложно предугадать – как именно следует атаковать определенное приложение в системе конкретного пользователя, даже если известно, что это приложение содержит уязвимость.

Впрочем, такое многообразие наблюдается на домашних машинах. В корпоративном секторе, да и среди мобильных устройств множество основных дистрибутивов и используемых продуктов очень сильно ограничено, и фактор разнообразия не очень актуален. В конце концов, существуют приложения, которые наверняка есть в любой системе — например, офисные пакеты и браузеры на десктопах или почтовые и веб-службы на серверах.

Последним аспектом, который хотелось бы затронуть, является связь общедоступности кода приложения и сложности организации атак на него. Встречается мнение, что закрытые приложения сложнее атаковать именно в силу невозможности получить доступ к исходному коду. Однако в случае атак на переполнение буфера, поиск уязвимостей посредством анализа исходного кода - трудоемкое занятие (если бы оно было простым или хорошо автоматизировалось, то потенциальные уязвимости легко бы выявлялись и исправлялись непосредственно разработчиками программ).

Атакующим гораздо проще отталкиваться от факта аварийного завершения приложения в некоторой ситуации. При «падении» приложения можно получить снимок образа его памяти и оценить — что привело к падению и можно ли это использовать в корыстных интересах. Последний шаг действительно проще сделать, имея доступ к исходному коду. Однако и анализ дизассемблированного кода программы не так сложен, как может показаться. И множество атак на проприетарные приложения — хорошее тому подтверждение.

Заключение

Таким образом, небольшая ошибка в небольшой программе может представлять серьезную угрозу для безопасности всей системы. Поэтому хотелось бы посоветовать разработчикам уделять должное внимание работе с памятью в своих продуктах.

Конечно, можно просто использовать языки программирования и соответствующие среды исполнения с автоматическим управлением памятью — такие как Java или Python, однако полностью заместить традиционные C/C++ они пока не могут.

Есть различные способы повысить безопасность системы и нивелировать ошибки в программах и на традиционных языках, однако разработчикам открытого ПО особо уповать на эти средства не стоит. Ведь их продукты распространяются преимущественно в виде исходного кода, и кто знает - с какими опциями он будет скомпилирован и на каком оборудовании будет запущен.

Пользователям же хочу посоветовать не быть пассивными и при обнаружении ошибок сегментации в приложениях (или просто при «падении» приложений в случае программ с графическим интерфейсом) сообщать об этом их авторам или разработчикам используемого дистрибутива. Ведь именно коллективная работа над ПО является силой Open Source, и пользователи — такие же участники этого процесса, как и разработчики.