B. L. Kozirsky, T. I. Komarov, M. A. Ivanov

Using Fuzz Testing for Searching Software Vulnerabilities

Keywords: software vulnerabilities, fuzzing, Black Box Testing

This article deals with fuzz testing (fuzzing), a software testing and vulnerability searching technique based on providing inputs of programs with random data and further analysis of their behavior. The basics of implementing cmdline argument fuzzer, environment variable fuzzer and syscall fuzzer in any UNIX-like OS have been closely investigated.

Б. Л. Козырский, Т. И. Комаров, М. А. Иванов

ИСПОЛЬЗОВАНИЕ ФАЗЗИНГА ДЛЯ ПОИСКА УЯЗВИМОСТЕЙ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ

Введение

Перед авторами данной статьи была поставлена задача поиска уязвимостей в ОС Эльбрус, применяющейся на вычислительных комплексах (ВК) семейства Эльбрус. Первоочередной проблемой стал выбор методов, использование которых позволило бы достичь наилучших результатов. Существующие методы поиска уязвимостей можно разделить на три большие группы: метод белого ящика; метод черного ящика; метод серого ящика. Метод белого ящика требует для своего использования исходных кодов ОС Эльбрус. Метод серого ящика теоретически может быть использован для исследования компонентов ОС Эльбрус, однако процессоры, используемые в ВК Эльбрус, имеют собственную уникальную архитектуру, которая недостаточно подробно описана в печатных работах и документации. Проведение реверс-инжиниринга исполняемых файлов, собранных для платформы Эльбрус, является слишком сложной и трудоемкой задачей. Метод черного ящика оказался наиболее подходящим для поиска уязвимостей в ОС Эльбрус, поскольку не требует знаний о внутреннем устройстве исследуемых объектов, а тестирование данным методом может быть легко автоматизировано. Фаззинг, который был выбран в качестве основного метода тестирования, представляет собой автоматическое тестирование методом черного ящика [1—5].

Основы фаззинга

Фаззинг — это метод обнаружения ошибок в ПО, в основе которого лежит следующая идея: на вход исследуемого приложения подаются самые разнообразные данные, а затем осуществляется анализ поведения исследуемого приложения. Для реализации поиска уязвимостей методом фаззинга на практике требуется, так или иначе, решить следующие задачи: определение цели; определение направлений ввода; подготовка данных; запуск программы с определенными данными; мониторинг исключений; исследование ошибок. Очевидно, что определение списка потенциальных целей и возможных направлений ввода производится разработчиками фаззера на этапе его создания. Например, если в качестве цели исследования были выбраны обычные программы, то в качестве направлений ввода могут выступать командная строка; переменные среды; данные, порождаемые другим ПО; имена и содержимое файлов и т. д. Подготовка данных, запуск тестируемой программы с определенными данными и мониторинг исключений — это основные функциональные возможности фаззера, которые должны быть реализованы. Не менее важным является и ведение логов, позволяющих быстро воспроизводить ситуации, приводящие к неожиданному поведению анализируемой программы. Исследование ошибок — это наиболее трудоемкий процесс, который практически невозможно автоматизировать. Фактически данный этап фаззинга представляет собой реверс-инжиниринг

тестируемого приложения, целью которого является детальный анализ ошибки и условий, при которых она может проявиться. Также исследователю необходимо ответить на вопрос, является ли найденная ошибка уязвимостью. На практике, однако, изучением и исправлением найденных ошибок занимаются разработчики ПО, а не специалисты, осуществляющие анализ защищенности ВК. В зависимости от того, какой объект выбран для исследования и какие направления ввода планируется использовать для проведения тестирования, можно выделить несколько типов фаззинга, а именно командной строки, переменной среды, формата файла, оперативной памяти, системных вызовов (локальные фаззеры), сетевых протоколов, веб-приложений и веб-браузеров (фаззеры удаленного доступа) (рис. 1). В рамках исследования защищенности ОС Эльбрус авторы проанализировали существующее и разработали собственное ПО, которое на практике реализует фаззинг командной строки (аргументов и опций), фаззинг переменной окружения и фаззинг системных вызовов.

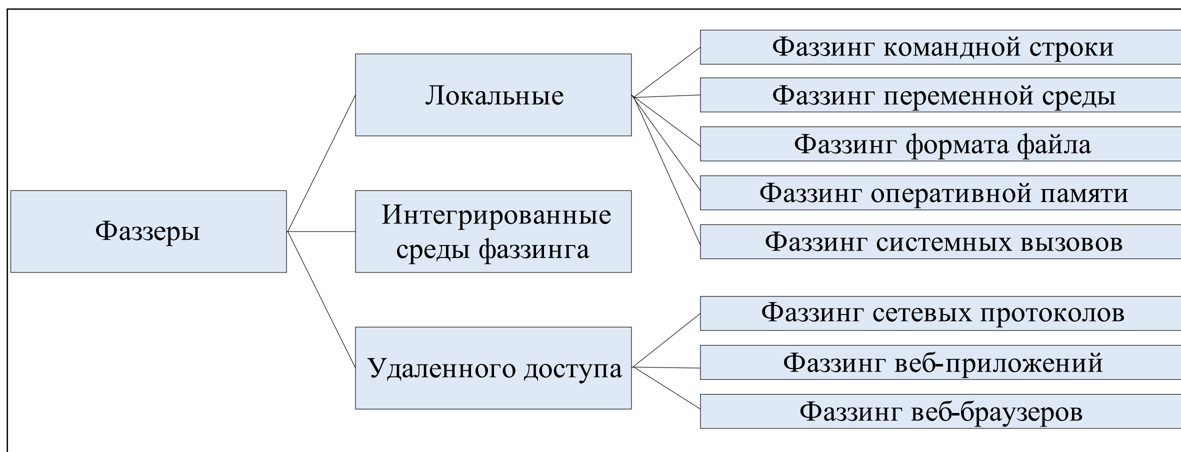

Рис. 1. Типы фаззинга

Фаззинг аргумента командной строки

Как правило, рядовой пользователь предпочитает программные продукты, управляемые при помощи графического интерфейса (GUI), но практически во всех UNIX-подобных ОС реализован интерфейс командной строки, который фактически является основным. Существуют различные способы ввода переменных в программу, среди которых можно выделить три основных: переменные среды окружения; ввод напрямую (чаще всего с клавиатуры); в качестве аргументов, с которыми вызывается программа. Рассмотрим третий способ более подробно. Внутри программы доступ к аргументам, переданным во время запуска, осуществляется при помощи переменной argv, имеющей тип char **. Также существует дополнительная переменная argc, которая содержит количество введенных аргументов, увеличенное на единицу (имя программы). Одной из задач программиста является корректная обработка этих переменных. Именно на данном этапе зачастую возникают серьезные ошибки, которые делают программу небезопасной. Идея такого вида тестирования заключается в том, что программа, получив в argc или argv какое-либо нестандартное значение, может продемонстрировать те или иные типы уязвимостей. Очевидно, что для злоумышленника, а следовательно, и для тестировщика наибольший интерес представляют привилегированные приложения. Но и программы, не обладающие какими-либо привилегиями, при неправильной работе могут создать серьезную угрозу для данных пользователя. В UNIX-подобных ОС привилегированные приложения найти довольно просто. Для этого достаточно найти приложения с установленными битами setuid или setgid, которые показывают, что при запуске программа выполнится с правами пользователя (либо группы), которым данный файл принадлежит. Для поиска подобных приложений можно воспользоваться командой find:

$find / -type f -perm -4001 -o -type f -perm -2001

Первый аргумент команды содержит в себе название каталога, с которого следует начинать рекурсивный поиск. Аргумент -type f указывает на то, что объектами поиска являются исключительно файлы. Флаг -perm показывает, что критерием поиска являются права доступа. Опции -4001 (-2001) обеспечивают поиск файлов, доступных на исполнение всем, притом с установленными битами setuid (setgid). Флаг -o позволяет использовать логику ИЛИ. Внимательный читатель может сказать, что данная команда не совсем верна, так как она найдет файлы с установленными setuid- или setgid-битами вне зависимости от того, какому пользователю они принадлежат. Разумеется, тестировщика интересуют лишь те пользователи, которые обладают особыми привилегиями. Такого рода поиск можно организовать, воспользовавшись флагом -user. В UNIX-подобных ОС подавляющее большинство программ принимают на вход аргументы в виде строк ASCII-символов. Такую строку легко сформировать при помощи интерпретатора языка Perl, поставляемого вместе с большинством дистрибутивов ОС:

```
$./target `perl –e 'print "A"x1000'`
```

Добавив в строку необходимое количество символов пробела, можно так же протестировать обработку argc. Для автоматизации фаззинга аргумента необходимо решить задачу определения некорректного поведения программы. В простейших случаях ОС демонстрирует это явно. Программа перестает отвечать, или выводится сообщение о какой-либо ошибке. Проблема заключается в том, что для автоматического фаззера такие признаки не годятся. Следовательно, нужно найти более надежные способы определения некорректной работы программы. Как правило, при обработке исключений и ошибок внутри программы инициируется завершение программы с каким-либо кодом, отличным от 0. В том случае, если программа завершилась при получении какого-либо сигнала, то код завершения будет равен 128 плюс номер сигнала. Так, например, ошибка сегментирования вызовет код 139 (сигнал SIGSEGV), а признак неверной инструкции — код 132 (сигнал SIGILL). Анализ кода завершения программы можно легко реализовать и использовать в автоматическом фаззере. В UNIX-подобной ОС фаззер предпочтительнее писать на языке программирования С. Во-первых, это обеспечивает высокую производительность, во-вторых, данный язык хорошо интегрирован в UNIX, его компиляторы присутствуют практически во всех дистрибутивах. Для реализации автоматического тестирования следует использовать связку функций fork() и wait(). Далее приведен пример кода с использованием вышеизложенного метода:

```
[…]
if ((pid = fork ()) != 0)
{
child = pid;
waitpid (pid, &status, 0);
  alarm (0);
   if (WIFSIGNALED (status))
          {
                    switch (WTERMSIG (status))
                    {
                              case SIGBUS:
                              case SIGILL:
                              case SIGSEGV:
                              case SIGTRAP:
                              case SIGFPE:
                              case SIGUSR1:
                              case SIGUSR2:
```

```
fprintf (stderr, " CRASH SIGNAL #%d\n", WTERMSIG (status));
                                        break;
                              default:
                                        break;
                    }
          }
}
else
{
 /*child */
  execle (fullpath, random_string(), "-h", "-z", "-zz", "----", NULL, environ);
  perror ("execle");
 }
[…]
```

От того, насколько разнообразные строки может генерировать функция random_string(), во многом зависит итоговая эффективность фаззинга. Пример реализации такой функции:

```
[…]
#define FS_TRIGGER "%n%n%n%n%n%n%n"
[…]
char *make_string (int size, char *startswith, char *endswith, char *hasone)
{
          char *buff;
          unsigned char filler;
          int min_size = 0;
          min_size =strlen (startswith) + strlen (endswith) + strlen (hasone) + 1;
          if (size < min_size)
          {
          printf ("error with the size of the string dude\n");
          exit (-1);
  }
          buff = malloc (size);
          if (!init)
  {
          srand (time ((time_t *) NULL) + getpid ());
          init = 1;
   }
          filler = (rand () % 254) + 1;
          if (hasone)
  {
          while ((strlen (hasone) == 1) && (filler == hasone[0]) && !(isprint (filler)))
                    filler = (rand () % 254) + 1;
  }
          memset (buff, filler, size);
          buff[size - 1] = 0x0;
          if (startswith)
  {
```

```
        memcpy (buff, startswith, strlen (startswith));
        memcpy (buff + strlen (startswith), FS_TRIGGER, strlen (FS_TRIGGER));
  }
        else
  {
        memcpy (buff + 1, FS_TRIGGER, strlen (FS_TRIGGER));
  }
        if (endswith)
  {
        strcpy (buff + strlen (buff) - strlen (endswith) - 1, endswith);
  }
        if (hasone)
  {
        memcpy (buff + (strlen (buff) / 2), hasone, strlen (hasone));
  }
        return buff;
}

char * random_string ()
{
char integer[2];
        int stringtype;
        int size;
        char *ptr;
        size = MAX_FUZZ_LENGTH;
        if (!init)
  {
        srand (time ((time_t *) NULL) + getpid ());
        init = 1;
  }
        stringtype = rand () % 18+1;
        switch (stringtype)
  {
        case 0:
              return make_string (size, "/", NULL, NULL);
        case 1:
              sprintf (integer, "%d", rand () % 9);
              return make_string (size, integer, NULL, NULL);
        case 2:
              return make_string (size, "#", NULL, NULL);
        case 3:
              return make_string (size, ".", NULL, NULL);
        case 4:
              return make_string (size, "-", NULL, NULL);
        case 5:
              return make_string (size, "--", NULL, NULL);
```

```
        case 6:
                return make_string (size, "%", NULL, NULL);
        case 7:
                return make_string (size, ":", NULL, NULL);
        case 8:
                return make_string (size, "+", NULL, NULL);
        case 9:
                return make_string (size, NULL, NULL, "/");
        case 10:
                return make_string (size, NULL, NULL, ":");
        case 11:
                return make_string (size, NULL, "/", NULL);
        case 12:
                return make_string (size, NULL, "%", NULL);
        case 13:
                return make_string (size, NULL, ":", NULL);
        case 14:
                return make_string (size, NULL, ":0", NULL);
        case 15:
                return make_string (size, "/", "/", NULL);
        case 16:
                return "/tmp";
        case 17:
                ptr = malloc (16);
                snprintf (ptr, 16 - 1, "%d", (rand () % 0xffffffff));
                return ptr;
                break;
  }
        return NULL;
}
[…]
```

Функция make_string() заполняет строку случайным ASCII-символом. Далее возможны несколько вариантов, какой именно — определяет функция random_string(). В начало необходимой строки может быть поставлен один из символов: «#», «.», «-», «--», «+», «%», «:». В другом варианте в середине строки содержится или «:», или «/». Третий вариант — строка заканчивается каким-либо из символов: «/», «%», «:», «:0». Данные символы помещаются в строку, чтобы сделать ее похожей на различные варианты реальных опций целевой программы. После первого символа строки помещается несколько символов «%n», это попытка обнаружить уязвимости вида «форматная строка». Разработанный фаззер, который использует данную методику, применялся авторами для поиска уязвимостей в ОС Эльбрус.

Фаззинг опций

Для выбора режима работы большинство программ в UNIX-подобных ОС принимают на вход различные опции (флаги). В участках кода, ответственных за обработку данных опций, могут содержаться ошибки, которые способны нарушить логику работы программы в целом. Для выявления таких ошибок можно воспользоваться фаззингом опций. Самым простым подвидом фаззинга опций является одноопциональный фаззинг. Он базируется на простом цикле, который проходит через

заранее заданный набор символов, используемых в качестве опций, передаваемых программе. При каждом новом вызове программе передается только одна опция. Однако можно предположить, что существуют такие уязвимости, которые могут быть выявлены только комбинацией различных опций. Чтобы получить новый тип фаззера, который способен комбинировать две опции, достаточно внутрь одноопционального фаззера поместить еще один одноопциональный фаззер. Таким образом, увеличение длины строки опций достигается добавлением нового одноопционального фаззера в одноопциональный фаззер предыдущего уровня. В общем случае тестируемая программа будет запускаться $N_1 \cdot N_2 \cdot \ldots \cdot N_n$ раз, где N_i — длина алфавита для каждой опции, n — длина строки опций. Фаззинг опций, как и фаззинг аргументов (argv и argc), обладает одним очень важным преимуществом — универсальностью. Другими словами, данный вид фаззинга всегда может быть использован — можно осуществлять тестирование сразу всего программного обеспечения в ОС, не «подгоняя» тестирующую программу под те или иные приложения. В противовес вышесказанному можно заметить, что задачей тестировщика может быть выявление ошибок и уязвимостей в какой-либо конкретной программе. Тогда для осуществления более эффективного тестирования следует учесть специфику целевого приложения. Такой подход может выявить уязвимости, недоступные для других видов тестирования. Для реализации данного подхода в случае с фаззингом опций следует внимательно изучить все опции, которые принимает программа. Простейший способ получить данные опции — просмотреть справку по программе, которую, как правило, можно получить при помощи ключа -help (-h). Так, например, выглядит справка по команде ping:

```
$ ping -h
Usage: ping [-LRUbdfnqrvVaAD] [-c count] [-i interval] [-w deadline]
   [-p pattern] [-s packetsize] [-t ttl] [-I interface]
   [-M pmtudisc-hint][-m mark] [-S sndbuf] [-T tstamp-options]
   [-Q tos] [hop1 ...] destination
```

Данный подход позволяет существенно повысить производительность фаззера, так как генерируемые опции не будут отброшены простейшими проверками внутри целевой программы. Также такой вид тестирования учитывает опции, состоящие более чем из одного символа. Авторами был написан на языке C собственный фаззер опций, при помощи которого осуществлялся поиск уязвимостей в привилегированных приложениях, входящих в состав дистрибутива ОС Эльбрус.

Фаззинг переменной среды

Данные могут передаваться в приложения через заранее предустановленные переменные среды окружения. Идея фаззинга переменной среды напоминает суть фаззинга аргументов командной строки, однако неожиданные входные данные, которые могут вскрыть уязвимости программы, помещаются в переменные среды. Простейший способ — протестировать приложение вручную. Предположим, что исследуемое приложение использует переменную PATH. Достаточно поместить в данную переменную окружения какую-либо неожиданную строку и проверить, как поведет себя программа:

```
PATH=` perl –e ‘print “a”x15000’` /usr/bin/target
```

Проблема состоит в том, что в большинстве случаев не известно, какие именно переменные среды окружения использует приложение. Для решения этой задачи можно воспользоваться отладчиком gdb. Необходимо запустить целевую программу в отладчике и поставить breakpoint на функцию getenv(). Так как отладчик не найдет объявление функции в целевой программе, то предложит поставить точку останова в подгружаемой разделяемой библиотеке, что, собственно говоря, и нужно. Далее следует задать команды, которые будут выполнены при достижении точки останова. Перед вызовом функции ее аргументы или ссылки на них будут помещены на стек или в соответствующие регистры, так что в упомянутых командах следует вывести на экран

содержимое области памяти, в которой хранится первый аргумент функции getenv(). Описанный выше способ приводит к удачному результату, но он является зависимым от аппаратной архитектуры. Существует другой способ фаззинга переменной среды, который не требует выяснять название каждой переменной среды, используемой приложением. Желаемого результата можно добиться, заставив функцию getenv() возвращать «неожиданную» строку всякий раз, когда данная функция вызывается приложением. Указанные возможности предоставляет механизм подгрузки библиотек. Механизм подгрузки библиотек работает следующим образом: пользователь помещает в определенную переменную среды путь до библиотеки, обладающей нужными ему функциональными возможностями. Если в данной библиотеке содержатся функции, совпадающие с функциями, используемыми в программе, то они будут использованы взамен исходных. Итак, требуется заменить стандартную системную функцию getenv() на функцию, возвращающую «неожиданную» строку, которая позволила бы выявить уязвимости, порожденные ошибками при обработке в программе переменных среды. Далее приведен простой пример кода, который решает данную задачу.

```
#define LENGTH 15000
char *getenv( char *variable)
{
        char buff[LENGTH];
        int i;
        for(i=0;i<LENGTH-1,++i)
buff[i]='A';
        buff[LENGTH -1] = 0x0;
        return buff;
}
```

После подгрузки библиотеки данный код будет использоваться вместо исходной функции getenv(). Для запуска целевой программы с использованием этого кода в качестве разделяемой библиотеки следует выполнить следующее:

```
cc –fPIC –o false_getenv.so false_getenv.c
LD_PRELOAD=./false_getenv.so /usr/bin/target
```

Метод, схожий с тем, что был описан выше, активно использовался авторами для анализа защищенности приложений, поставляемых в дистрибутиве ОС Эльбрус, и позволил обнаружить ряд небольших уязвимостей, связанных с некорректной обработкой переменных среды окружения.

Фаззинг системных вызовов

Данный тип фаззинга предназначен для поиска уязвимостей в ядре ОС, он осуществляется путем запуска системных вызовов с использованием случайных данных в качестве их аргументов. Главная проблема этого метода фаззинга состоит в том, что многие системные вызовы осуществляют проверку корректности переданных аргументов. Следовательно, для эффективного тестирования требуется привнесение в данный метод фаззинга некой интеллектуальности. Например, если системный вызов ожидает получения в качестве аргумента данных определенного типа, то он должен получить именно их. Таким образом, фаззер должен создавать все условия для корректного запуска системных вызовов. Данный подход существенно повышает эффективность тестирования по сравнению с более грубым подходом, подразумевающим запуск системных вызовов с абсолютно случайными аргументами. Однако обеспечение таких функциональных возможностей требует очень серьезного усложнения кода, а также лишает фаззер совместимости с другими ОС и аппаратными платформами. Еще одна проблема, требующая решения, — фиксирование того факта,

что приложение ведет себя недокументированным способом. Часто это не является тривиальной задачей. Простейшее решение данной проблемы — проверка возвращаемого приложением кода. Тестирование системных вызовов также требует анализа возвращаемых значений, поэтому любой фаззер системных вызовов должен иметь возможность ведения логов, достаточно информативных для подробного изучения тестировщиком и ручного воспроизведения возникающих в ходе работы фаззера событий. Для реализации фаззера системных вызовов в первую очередь следует составить список системных вызовов в удобном для применения в программе виде. Для составления данного списка можно использовать следующие источники информации: документацию, заголовочные файлы */usr/include/bits/syscall.h* и */usr/include/asm/unistd.h*. Авторы разработали собственный фаззер системных вызовов, адаптированный для применения в ОС Эльбрус, который, однако, может быть легко портирован для использования на других платформах. Вновь в качестве основного языка разработки был выбран C. Для хранения списка системных вызовов был использован массив структур, в котором номер структуры соответствует номеру вызова в системе. Данное решение позволяет легко извлекать имя системного вызова, обеспечивает возможность быстрого добавления флагов системного вызова и новых системных вызовов. Разработанный фаззер имеет два режима работы. Первый режим работы реализует фаззинг, базирующийся на использовании полностью случайных данных. Номер осуществляемого вызова определяется генератором псевдослучайных чисел, с его же помощью заполняются процессорные регистры. Второй режим работы созданного фаззера — это режим ротации. Определенное числовое значение проходит через все процессорные регистры во всех возможных комбинациях. Выглядит это примерно следующим образом:

<---->, <---->, <---->, <---->, <---->, VALUE
<---->, <---->, <---->, <---->, VALUE, <---->
<---->, <---->, <---->, <---->, VALUE, VALUE
<---->, <---->, <---->, VALUE, <---->, <---->

В качестве VALUE, например, может быть использовано значение по умолчанию (0), адрес из userspace или адрес, по которому загружено ядро. В разработанную программу были также добавлены некоторые дополнительные опции, не имеющие непосредственного отношения к режимам работы фаззера. Например, реализована возможность тестирования одного конкретного вызова или возможность остановить тестирование после выполнения определенного количества вызовов, что весьма актуально, так как даже после небольшого по времени тестирования логи имеют весьма внушительный размер, что существенно затрудняет анализ.

Кратко рассмотрим порядок работы фаззера. Первый этап его работы — это инициализация. На данном этапе удаляются файлы логов, оставшиеся после предыдущего сеанса работы фаззера; определяется размер страницы памяти; создается так называемый пользовательский буфер, который состоит из трех секций, по 4 килобайта каждая. Все байты первой и последней секции заполняются предопределенной константой, байты средней секции заполняются нулями. Структура буфера представлена на рис. 2.

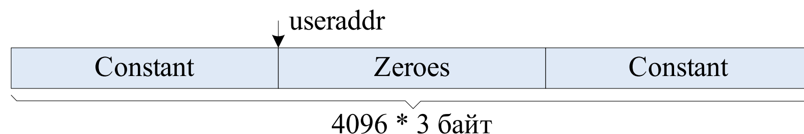

Рис. 2. Структура буфера

На этом же этапе работы определяется useraddr = userbuffer + 4096. Данная величина используется в качестве адреса из userspace, упомянутого в предыдущем разделе. При осуществлении

фаззинга проверяется, не произошло ли повреждение областей памяти, соседних с той, адрес которой передается в качестве аргумента системным вызовам. Далее устанавливается реакция фаззера на поступающие сигналы. Основная задача — минимизировать воздействие сигналов на процесс тестирования. Поступающие сигналы должны лишь информировать пользователя о необычном поведении системы, не мешая тестированию. Это достигается использованием общеизвестной функции sigaction. С ее помощью устанавливается обработчик сигнала, который выводит имя поступившего сигнала на экран, устанавливается флаг SA_RESTART, который позволяет работать некоторым системным вызовам во время обработки сигналов. Также для каждого сигнала на время работы его обработчика маскируются все прочие сигналы. Запуск системных вызовов осуществляется в дочернем процессе, а то, что с ним происходит, представляет слабый интерес, поэтому сигнал SIGCHLD игнорируется. Исключением является сигнал SIGALRM. Для него используется стандартный обработчик, так как необходимо ограничивать время, за которое выполняется один системный вызов. Дальнейший ход инициализации напрямую зависит от переданных программе аргументов. Разбор переданной пользователем строки осуществляется при помощи стандартной функции getopt_long. В зависимости от переданных опций устанавливаются переменные, определяющие дальнейший режим работы. На этом завершается первый этап работы фаззера. Следующим этапом работы является непосредственное тестирование системы. В основе данного этапа лежит бесконечный цикл. Операции, выполняемые в теле данного цикла, зависят от выбранного режима работы фаззера. В качестве примера рассмотрим первый режим работы фаззера. В теле цикла происходит вызов функции, внутри которой, в свою очередь, происходит порождение нового процесса. Внутри этого процесса осуществляется проверка, на предмет того, происходит ли тестирование конкретного системного вызова, или осуществляется полное тестирование всех вызовов, присутствующих в системе. В последнем случае происходит генерация номера системного вызова при помощи функции rand по модулю NR_SYSCALLS (количество вызовов в системе). Затем осуществляется проверка флагов, хранящихся в той же структуре, что и имя осуществляемого вызова. Далее запускается таймер alarm, на генерацию аргументов и на выполнение самого вызова выделяется не более 3 секунд. Именно для этой цели механизм обработки сигнала SIGALRM был оставлен без изменений. Потом производится генерация аргументов, в ходе которой каждый процессорный регистр заполняется при помощи функции rand, и, наконец, выполняется сам системный вызов при помощи функции syscall. После вызова проверяется код возврата и, если он меньше нуля, выводится код ошибки. В том случае, если вызов вернул ENOSYS (это значит, что системный вызов не реализован), в структуре, хранящей информацию о данном вызове, устанавливается флаг, обозначающий, что в дальнейшем этот вызов выполнять не следует. Далее следует вспомнить о буфере, про инициализацию которого говорилось в начале раздела. В данный момент осуществляется проверка содержимого этого буфера. Если после вызова оно изменилось, то выводится сообщение, информирующее пользователя об этом. Выполнив все эти операции, дочерний процесс завершается. Далее, уже в теле основного цикла осуществляется проверка, не превышен ли лимит количества вызовов, если таковой был установлен. Если лимит превышен, то происходит выход из цикла и завершение тестирования.

Тестирование ВК Эльбрус методом фаззинга системных вызовов длилось весьма продолжительное время, с высокой степенью уверенности можно заявить, что системные вызовы, реализованные в ОС Эльбрус, достаточно стабильны. Тем не менее авторам удалось обнаружить два системных вызова, реализация которых содержит ошибки, которые могут приводить к нестабильному поведению ядра ОС.

Заключение

В статье рассмотрены принципы построения фаззеров аргументов командной строки, переменных окружения и системных вызовов. Данные способы поиска уязвимостей доказали свою

эффективность при анализе защищенности ОС Эльбрус. Авторы убеждены, что исследования в данной области должны быть продолжены, поскольку существуют направления фаззинга, которые являются недостаточно изученными и проверенными на практике. В качестве примеров подобных направлений можно привести фаззинг оперативной памяти или фаззинг сетевых протоколов. Данные типы фаззинга, наряду с уже рассмотренными, могут быть применены для исследования защищенности фактически любых ОС.

СПИСОК ЛИТЕРАТУРЫ:

1. *Erickson J*. Hacking: The Art of Exploitation. 2nd edition. No Starch Press. 2008.
2. Oulu University Secure Programming Group [Электронный ресурс]. URL: https://www.ee.oulu.fi/research/ouspg/ (дата обращения: 22.09.2014).
3. *Sutton M., Green A., Amini P*. Fuzzing. Brute Force Vulnerability Discovery. Addison-Wesley. 2007.
4. Offensive Security [Электронный ресурс]. URL: http://www.offensive-security.com (дата обращения: 25.10.2014).
5. Blog about system and low-level programming [Электронный ресурс]. URL: http://www.syprog.blogpspot.ru (дата обращения: 02.05.2013).

REFERENCES:

1. *Erickson J*. Hacking: The Art of Exploitation. 2nd edition. No Starch Press. 2008.
2. Oulu University Secure Programming Group. URL: https://www.ee.oulu.fi/research/ouspg/.
3. *Sutton M., Green A., Amini P*. Fuzzing. Brute Force Vulnerability Discovery. Addison-Wesley. 2007.
4. Offensive Security. URL: http://www.offensive-security.com.
5. Blog about system and low-level programming. URL: http://www.syprog.blogpspot.ru.