УДК 004.056

# Разработка методики применения фаззинга для анализа уязвимостей программного обеспечения

Томилов И. О., Карманов И. Н.,
Звягинцева П. А., Грицкевич Е. В.

***Актуальность и постановка задачи:*** *в настоящее время информационные ресурсы становятся главной целью атак со стороны злоумышленников. Атакующая сторона активно использует наиболее распространенные уязвимости систем: утечки памяти, недостатки шифрования, переполнение буфера, неполную проверку входных данных и т.д. Атакуемая сторона, в свою очередь, старается находить уязвимые места системы, разрабатывая и совершенствуя барьеры, ограждающие защищаемый информационный объект от несанкционированного доступа. Для защиты программно-информационных ресурсов часто используются стандартные методики обеспечения информационной безопасности системы и ее программного обеспечения. Наиболее распространённый подход к обеспечению безопасности - это периодическое обновление системы и приложений. Обновление закрывает уязвимости и исправляет ошибки, которые были допущены в программах в момент их создания. Однако, при этом не учитывается актуальность угроз и реальная востребованность конкретного обновления. Подобная концепция не обеспечивает комплексного подхода к безопасности.* ***Целью работы*** *является разработка системы фаззинг-тестирования программного обеспечения на предмет наличия в нем уязвимостей с включением в разрабатываемую систему так называемого интеллектуального агента.* ***Объектом исследования*** *выступает процесс обнаружения уязвимостей в программном обеспечении.* ***Предметом исследования*** *являются технологии фаззинга, применяемые во время поиска уязвимостей.* ***Используемые методы:*** *решением проблемы является динамический анализ процесса выполнения программного кода. Преимущество такого подхода состоит в том, что при данном тестировании практически полностью отсутствуют ложные срабатывания, в отличие от статических анализаторов.* ***Новизна:*** *элементами практической новизны являются определение требований к системе тестирования и разработка алгоритма и методики фаззинг-тестирования.* ***Результат:*** *разработанный алгоритм и скрипт для проведения фаззинг-тестирования показал свою работоспособность при нахождении уязвимостей в экспериментальном исследовании известных общедоступных программ.* ***Практическая значимость:*** *обнаружение даже одной уязвимости за короткий срок в автоматическом режиме можно считать свидетельством надежности технологии, позволяющей достаточно эффективно блокировать возможные атаки злоумышленников.*

***Ключевые слова:*** *фаззинг, уязвимости программного обеспечения, тестирование программного обеспечения, киберугрозы, аудит программного обеспечения, интеллектуальный агент.*

## Постановка задачи

По мере развития и усложнения информационных систем обостряются проблемы их информационной безопасности. Этот факт предъявляет особые требования к своевременному обнаружению уязвимостей используемого про-

---

**Библиографическая ссылка на статью:**
Томилов И. О., Карманов И. Н., Звягинцева П. А., Грицкевич Е. В. Разработка методики применения фаззинга для анализа уязвимостей программного обеспечения // Системы управления, связи и безопасности. 2018. № 4. С. 48-63. URL: http://sccs.intelgr.com/archive/2018-04/03-Tomilov.pdf.

**Reference for citation:**
Tomilov I. O., Karmanov I. N., Zvyagintseva P. A., Gritskevich E. V. Development of fuzzing application technique for software vulnerabilities analysis. *Systems of Control, Communication and Security*, 2018, no. 4, pp. 48-63. Available at: http://sccs.intelgr.com/archive/2018-04/03-Tomilov.pdf (in Russian).

граммного обеспечения на этапах его разработки, создания информационной инфраструктуры и проведения аудита готовой инфраструктуры.

Для этих целей используется тестирование программ, при котором программы анализируются на предмет способности противостоять нежелательному внешнему вмешательству в их работу. Для тестирования применяются как статический анализ на этапе создания программного кода, так и динамический анализ в процессе выполнения программы. Для динамического анализа используется фаззинг, который является технологией тестирования программ в автоматическом режиме с целью выявления потенциальных уязвимостей. При таком тестировании практически полностью отсутствуют ложные срабатывания, характерные для статических анализаторов. При этом обеспечивается большое количество граничных значений путём создания некорректных входных данных. Входными данными выступают обрабатываемые исследуемым приложением файлы и другая информация, в том числе определяемая протоколами обмена, прикладными интерфейсными функциями и т.п.

В настоящей работе рассматривается тестирование программного обеспечения по технологии фаззинг, для чего была произведена разработка системы фаззинг-тестирования с включением в неё так называемого интеллектуального агента, а затем проведено ее экспериментальное исследование. При этом решались следующие задачи: определить набор требований к фаззеру; разработать фаззер, включающий в себя комплексные решения тестирования; протестировать полученное решение.

## Анализ технологии «фаззинг» как средства проведения аудита программного обеспечения

В настоящее время технология «фаззинг» весьма востребована в информационном сообществе, поскольку позволяет выявить уязвимости в приложениях и системах прежде, чем ими успеют воспользоваться злоумышленники. Фаззинг представляет собой автоматическое тестирование программного обеспечения, заключающееся в том, что на вход исследуемой программы подаются случайные, модифицированные, неправильные значения, вызывая аномалии в поведении программы.

В зависимости от методики генерирования входной информации существующие технологии фазинга можно условно разделить на два класса:

1) мутация существующих данных;
2) генерация новых данных.

Ниже приведены типы фаззинга, каждый из которых относится к одному из этих классов.

Тип 1. Заранее подготовленные данные.

Тип 2. Использование случайных данных. Является наименее эффективным подходом, поскольку определение причины возникшего сбоя в этом случае весьма затруднительно.

Тип 3. Ручное изменение данных. Путем внесения заведомо ошибочной входной информации исследователь искусственно добивается сбоя работы

анализируемой программы. Эффективность такого подхода мала по причине отсутствия автоматизации процесса.

Тип 4. Перебор мутаций данных. При этом объем тестов уменьшается, но, в то же время, объем данных сильно увеличивается, поскольку данные подвергаются значительным мутациям.

Тип 5. Осознанное внесение изменений в данные. Для проведения такого тестирования осуществляется дополнительный анализ с целью определения данных, которые не будут изменяться, и данных, которые, напротив, подлежат изменениям. Такой подход требует больших временных затрат.

В настоящее время имеется большое количество специализированной литературы, посвященной, как подробному анализу различных методов фаззинг-тестирования, так и анализу уязвимостей программного обеспечения, которые фаззинг позволяет выявить (см., например, [1-7]). В этих источниках, как правило, исходные коды тестирующих программ если и приводятся, то крайне фрагментарно, что не дает возможности их практического применения. В данной работе делается попытка более подробно проследить процесс разработки фаззера от этапа определения требований к тестирующей системе до ее экспериментального исследования.

## Определение требований к системе тестирования и разработка алгоритма функционирования фаззера

Набор ключевых компонентов разрабатываемой системы тестирования (фаззера) и их функций был определен исходя из предъявляемых к системе требований: автоматизация работы, то есть автоматический контроль за состоянием исследуемого процесса (с его перезапуском в случае сбоя) и автоматическая регистрация результатов работы фаззера (включая отладочную информацию); централизованное управление и централизованное хранение результатов работы.

Упрощённый алгоритм, реализующий методику применения фаззера для анализа уязвимостей программного обеспечения, приведен на рис. 1.

Разрабатываемая система фаззинга включает в себя так называемый интеллектуальный агент (intelligent agent) [8]. Программа, которая содержит в своем составе интеллектуальный агент – это система, самостоятельно выполняющая указанное пользователем задание в течение длительных промежутков времени. В данном случае термин «интеллектуальный» подчеркивает более высокий уровень технологии управления по сравнению с примитивными (*dumb*) триггерными системами автоматического управления. Это может быть как программной системой, так и сложной автоматизированной системой.

Функции интеллектуального агента могут быть следующими:

- самостоятельное выполнение задания, указанного пользователем;
- автоматическое распознавание требуемой задачи и необходимых для нее инструментов;
- запуск требуемых задач в определенное пользователем время.

Схема функционирования интеллектуального агента представлена на рис. 2.

Рис. 1. Упрощённый алгоритм работы фаззера

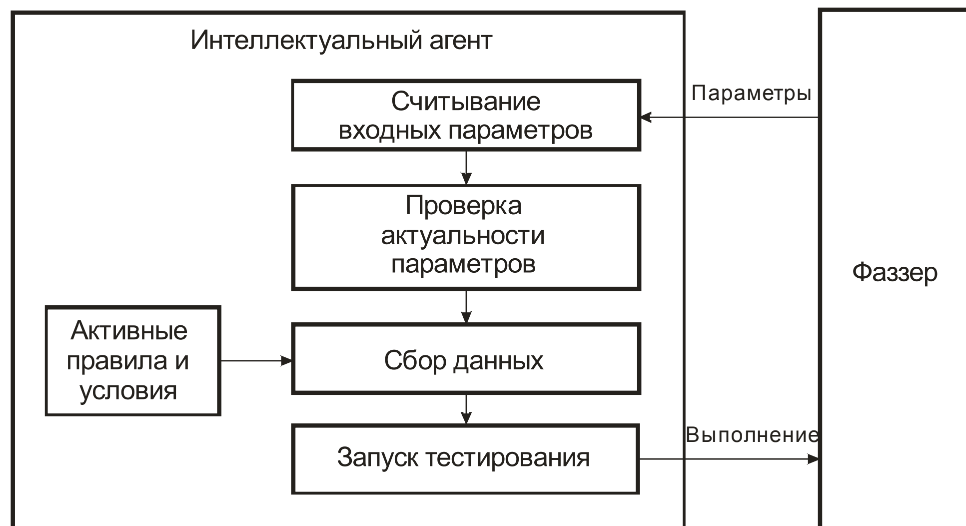


Рис. 2. Схема функционирования интеллектуального агента

В разрабатываемой системе задействованы вышеуказанные функции, а также задействован целенаправленный агент. Целенаправленный агент хранит информацию о тех ситуациях, которые для него желательны. Это дает агенту способ выбрать среди многих путей тот, который приведет к нужной цели.

### Разработка системы фаззинг-тестирования

Тестирование требует организации большого количества итераций воздействия входными данными. При этом точка воздействия единая, и состояние программы необходимо возвращать в исходное после каждой итерации. Фаззеры, действующие на программу через файлы, сокеты, перемененные окружения пользуются тем фактом, что программа после синхронной или асинхронной обработки входных данных снова готова принимать входные данные. При этом, в общем случае, нельзя гарантировать, что на результаты воздействия текущей итерации не повлияли данные из предыдущих итераций. Однако, с некоторыми допущениями, можно считать, что это так.

Недостатком воздействия через стандартный ввод является сложность достижения внутренних функций, так как на пути к ним препятствием становится ряд условий интерфейсных функций. Решением проблемы является организация воздействия на внутренние функции, минуя интерфейсные. Стандартный ввод в этом случае не позволяет реализовать нужную схему. Метод «фаззинга в памяти» [1] предоставляет такую возможность. Использование цикла мутации или восстановления состояния делает возможным итеративное воздействие на выбранный участок кода. Восстановление состояния решает проблему влияния предыдущих итераций, однако, является достаточно трудоемкой процедурой с привлечением больших ресурсов.

Изолирование тестируемого участка кода – это одна из техник модульного тестирования. Применяемый при разработке системы фаззинг-тестирования принцип модульного тестирования наглядно поясняется на рис. 3.

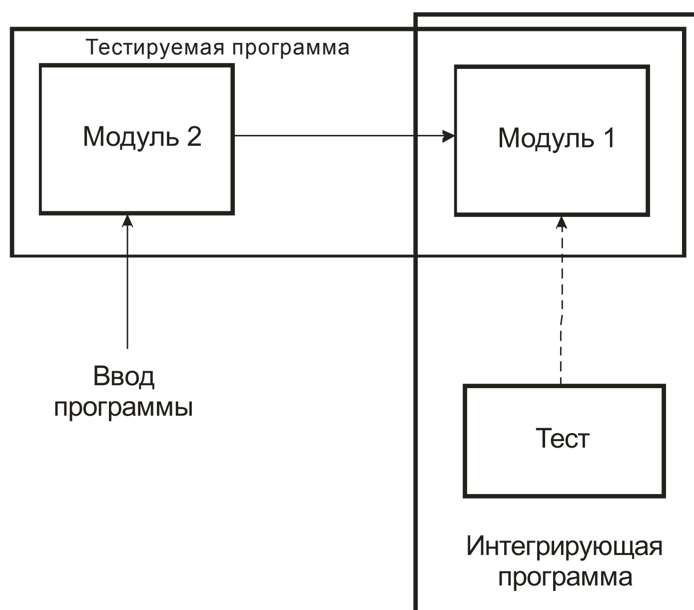


Рис. 3. Принцип модульного тестирования

На рис. 3 представлена ситуация, когда программа (изображена как горизонтальный прямоугольник) состоит из двух модулей: модуль 1 и модуль 2. При этом интерфейсным является модуль 2. Функции модуля 1 достижимы через ввод программы, обрабатываемый модулем 2. Таким образом, тестировать

модуль 1 можно передавая данные модулю 2, однако, более эффективной будет передача данных непосредственно модулю 1.

На этапе разработки тестирование может производиться на отдельных модулях. В этом случае тестирующий модуль вызывает функции тестируемого модуля (которые в данной ситуации являются интерфейсными). На исполнение запускается интегрирующая программа – результат связывания (статического) модуля 1 и тестового модуля (эта программа изображена в виде вертикального прямоугольника).

При использовании модульных тестов доступны следующие преимущества:

- возможность воздействия на выбранную функцию;
- возможность организации цикла воздействия;
- минимальное искажение исследуемой программы.

Применение схемы на этапе верификации, однако, требует решения некоторых технических проблем:

- определения точки воздействия, поскольку в отсутствие исходных кодов точка воздействия выбирается либо в ходе ручного анализа исполняемого кода, либо автоматически в соответствии с заданным критерием;
- организации интерфейсов для выбранной точки внедрения данных;
- динамического связывания тестирующего модуля с тестируемым.

Помимо данных о степени готовности тестирования программы важно также иметь представление о том, какой наилучший результат тестирования может быть достигнут при воздействии на определенную точку программы. Статический и динамический анализ исследуемой программы позволяет получить достаточно данных, чтобы оценить перспективность выбора той или иной точки внедрения. Здесь не рассматривается внедрение данных в точки, отличные от начала функций. Функции удобнее для организации воздействия, потому что они имеют вполне определенный формат ввода (прототип). Вопрос восстановления прототипов функций из исполняемого кода рассмотрен ниже.

Функция более перспективна для внедрения воздействий, если, следуя по всем возможным путям исполнения программы, начиная с данной функции, можно достигнуть наибольшего количества ветвей. Задача оценки функций по данному критерию сводится к решению задачи достижимости остальных функций, начиная с данной. Следует учитывать, что не все функции одинаково удобны для внедрения воздействий. Более удобными являются функции, принимающие в качестве параметров строки или указатели на массивы. Таким образом, после автоматической оценки перспективности функций должен следовать этап фильтрации наиболее перспективных функций экспертом. Данный процесс упростит статистика передаваемых в функцию данных во время динамического анализа.

Для обеспечения наибольшего покрытия кода необходим массивный набор входных данных. В данной работе применялись следующие подходы подготовки данных:

- генерация данных, случайная или в соответствии со спецификацией;

- мутация данных существующих образцов, представляющая из себя случайное или заданное правилами изменение отдельных частей образца.

Для генерации образцов требуется знание спецификации обрабатываемых приложением данных. Подготовка генератора – задача трудоемкая. Однако, полученные таким образом данные будут использовать большинство возможностей спецификации, а значит и покрывать больший объем кода, обрабатывающего данные. Мутация данных – более простой в реализации подход, однако он требует большого набора начальных образцов, которые затем изменяются.

В качестве входных данных фаззер принимает исполняемые файлы исследуемого программного обеспечения. Исходный код и спецификация исследуемого программного обеспечения не требуются.

Выходными данными метода являются: информация о найденных сбоях; входные данные, которые приводят к сбою; графы вызовов и взаимосвязей линейных блоков функций. Выходные данные достаточны для проведения дальнейшего автоматизированного анализа уязвимостей.

Для повышения охвата тестируемых систем были выбраны, улучшены и объединены следующие имеющиеся решения [8]:

- XSStrike – это расширенный набор обнаружения XSS, который обеспечивает нулевые ложноположительные результаты с использованием нечеткого соответствия, и имеет широкую базу пэйлоадов;
- Aggroargs – брутфорс переполнения буфера командной строки linux;
- Backfuzz – набор для фаззинга сетевых протоколов;
- Brutexss – брутфорс межсайтового скриптинга;
- Cirt-fuzzer – набор для фаззинга протоколов TCP/UDP;
- Crlf-injector – скрипт для тестирования проблем, вызванных символом;
- CRLF (перевод строки);
- Dhcpig – набор для фаззинга DHCPv4 и DHCPv6
- Fuzztalk – набор для фаззинга XML;
- Http-fuzz – набор для фаззинга http.

Современное программное обеспечение имеет, как правило, модульную архитектуру. Модульность обуславливает как практику разделения функциональности с целью повторного использования кода при проектировании, так и особенности взаимодействия с операционной системой. С целью разделения общих ресурсов и кода между различными программами применяются динамические библиотеки, механизм использования которых имеется практически во всех современных операционных системах. Таким образом, при выборе цели нужно определиться не только с тестируемым приложением, но и с модулями, которые представляют интерес. Это позволит сфокусироваться на исследовании интересующего кода. В программной реализации метода список интересующих модулей задается в специальном конфигурационном файле. Для отмеченных модулей сразу же после загрузки выполняется статический анализ.

Непосредственно при разработке тестирующей системы на первом этапе был написан набор для тестирования на служебные символы. Служебные сим-

волы – это скобки, запятые, двоеточия, разделители (пробелы). Благодаря этому в разных местах будет нарушаться структура корректного запроса:

```
base='{"example" : {"innerobj" : "someval"}, "example" : 777777777, "exam-
ple" : [1, [2, {"inlist" : "val"}], 3], "end" : "543"}'

ad1='Access Denied, pass tag not found in JSON..'
ad2='Exit code = 0'

if1='Incorrect data format! Check your JSON syntax.'
if2='Exit code = 1'
declare -a checkable_syms=('[' ']' '{' '}' ' ' ':' ',')
declare -a arr=(" " "]" "{" "[[" "}}" ":" "," "A" "1" ";")
echo "Fuzzing maintenance symbols.."
for symbol in "${checkable_syms[@]}"
do
num=$(($(echo $base | awk "BEGIN{FS=\"[$symbol]\"} {print NF}") - 1))

      for i in $(seq 1 $num)
      do
            for bad_sym in "${arr[@]}"
            do
                  if [[ "$bad_sym" != "$symbol" ]]; then
                        [[ (("$resp" =~ "$if1" && "$resp" =~ "$if2")) ||
(("$resp" =~ "$ad1" && "$resp" =~ "$ad2")) ]] || echo $resp
                  fi
            done
      done
done
```

Затем производилось тестирование уровня вложенности объектов друг в друга:

```
N=1024
base='"{\"example\" : "'
final='"{\"innerobj\" : \"someval\"}"'
ad1='Access Denied, pass tag not found in '
ad2='Exit code = 0'
if1='Incorrect data format! Check your syntax.'
if2='Exit code = 1'
echo "Fuzzing nested objects.."
for i in $(seq 1 $N)
do
      que="$base*$i + $final + \"}\"*$i + \"\x00\""
      pyt="print($que);"

      [[ (("$resp" =~ "$ad1" && "$resp" =~ "$ad2")) ]] || echo $resp
Done
```

Тестирование множественных запросов было реализовано следующим образом:

```
N=2048
base1='{\"example'
letter='A'
final1='\": \"example\"}'
base2='{\"example\" : \"example'
final2='\"}'
ad1='Access Denied, pass tag not found in.'
ad2='Exit code = 0'
if1='Incorrect data format! Check your syntax.'
if2='Exit code = 1'
```

```
flag=0
echo "Fuzzing long strings.."
for i in $(seq 1 $N)
do
      if [[ "$flag" == 0 ]]; then
            base=$base1
            final=$final1
            flag=1
      else
            base=$base2
            final=$final2
            flag=0
      fi
      que="\"$base\" + (\"$letter\")*$i + \"$final\" + \"\x00\""
      pyt="print($que);"
      [[ (("$resp" =~ "$ad1" && "$resp" =~ "$ad2")) ]] || echo $resp
done
```

Ниже приведен программный фрагмент, описывающий тело фаззера:

```
def fuzzer(url, param_data, GET, POST):
    result = []
    progress = 0
    for i in fuzzes:
        progress = progress + 1
        sleep(delay)
        sys.stdout.write('\r%s Fuzz Sent: %i/%i' % (run, progress,
len(fuzzes)))
        sys.stdout.flush()
        fuzzy = quote_plus(i)
param_data_injected = param_data.replace(xsschecker, fuzzy) if GET:
r = requests.get(url + param_data_injected) # makes a request to
example.search.php?q=<fuzz>
        else:
            r = requests.post(url, data=param_data_injected) # Seperating
"param_data_injected" with comma because its POST data
        response = r.text
        if i in response:            result.append({
            'result' : '%sWorks%s' % (green, end),
            'fuzz' : i})
        elif str(r.status_code)[:1] != '2'            result.append({
            'result' : '%sBlocked%s'  % (red, end),
            'fuzz' : i})
        else:            result.append({
                'result' : '%sFiltered%s' % (yellow, end),
                'fuzz' : i})

    table = PrettyTable(['Fuzz', 'Response'])
        table.add_row([value['fuzz'], value['result']])
print('\n', table)
```

Полный код написанного фаззера здесь не приводится. При экспериментальном исследовании фаззера проводился анализ уязвимостей на основе данных о сбое. Сбой в работе тестируемой программы не всегда является уязвимостью безопасности. В общем случае программный сбой относится к качеству программного обеспечения.

Сбой становится уязвимостью, когда он создает одну из следующих возможностей:

- удаленно выполнить код;
- инициировать отказ обслуживания;
- обойти заданную политику безопасности (например, повысить привилегии).

### Экспериментальное исследование фаззера

Экспериментальное исследование фаззера проводилось на известных общедоступных программах. Так, например, была протестирована антивирусная программа Trend Micro Titanium Maximum Security [9].

Вскоре после запуска фаззера происходит аварийное завершение работы системы. В выводе удаленного отладчика режима ядра отображается сообщение с информацией о последнем обработанном фаззером IOCTL-запросе:

```
'C:\Program Files\Trend Micro\AMSP\coreServiceShell.exe' (PID: 792)
'\Device\TmComm' (0x81d6a030) [\SystemRoot\system32\DRIVERS\tmcomm.sys
IOCTL Code: 0x9000402b,  Method: METHOD_NEITHER
InBuff: 0x018fc080,  InSize: 0x0000004c
OutBuff: 0x018fc080, OutSize: 0x0000004c
```

В данном фрагменте фигурируют имена драйвера и процесса Trend Micro. Далее был произведен реверсинг уязвимого драйвера *tmcomm.sys*, начиная с процедуры обработки IRP-запросов к устройствам данного драйвера:

```
int __stdcall sub_1E8BE(int DriverObject, char *Irp)
{
StackLocation = *((_DWORD *)Irp + 24);
IoStatus = Irp + 28;
*((_DWORD *)Irp + 7) = 0;
MajorFunction = *(_BYTE *)StackLocation;
if (MajorFunction == 2)
goto LABEL_12;
if (MajorFunction > 0xDu)
{
// обработка IRP запросов типа IRP_MJ_DEVICE_CONTROL
if (MajorFunction <= 0xFu)
{
// извлечение параметров IOCTL запроса из структуры IO_STACK_LOCATION
      Type3InputBuffer = *(_DWORD *)(StackLocation + 0x10);
      UserBuffer = *((_DWORD *)v6 + 15);
      InputBufferLength = *(_DWORD *)(StackLocation + 8);
      OutputBufferLength = *(_DWORD *)(StackLocation + 4);
      v27 = IoStatus;
      v24 = OutputBufferLength;
// дальнейшая обработка IOCTL запроса
      v5 = sub_1F7A6(*(_DWORD *)(StackLocation + 0xC),
(int)&InputBufferLength);
      goto LABEL_9;
 }
// ...
}
// ...
return v5;
}
```

Как видно по приведенному псевдокоду, обработка IOCTL-запросов осуществляется в процедуре sub_1F7A6():

```
int __stdcall sub_1F7A6(int ControlCode, int a2)
{
 v7 = 0xC00000BBu;
 v6 = 0;
 v2 = ExGetPreviousMode();
 v3 = 0;
 if (off_34CB4)
 {
 v4 = 0;
// Поиск процедуры дальнейшей обработки IOCTL запроса по значению
// ControlCode. Для 0x9000402b (значение, которое было выявленно при фаз-
зинге)
// будет вызвана процедура sub_1FF38()
while (*(int *)((char *)&dword_34CB0 + v4) != ControlCode)
 {
       ++v3;
       v4 = 8 * v3;
       if (!*(&off_34CB4 + 2 * v3))
       goto LABEL_7;
 }
 v6 = *(&off_34CB4 + 2 * v3);
}
LABEL_7:
 if (v2 == UserMode)
 {
// проверка входного и выходного буферов
ProbeForRead(*(const void **)(a2 + 8), *(_DWORD *)a2, 1u);
ProbeForWrite(*(PVOID *)(a2 + 12), *(_DWORD *)(a2 + 4), 1u);
}
```

Полный код всех процедур, которые участвуют в обработке IOCTL-запроса здесь не приводится.

В ходе проведенного реверсинга было выяснено, что IOCTL-запрос с кодом 0x9000402b используется для вызова из пользовательского приложения оригинальных обработчиков тех системных вызовов, которые были перехвачены драйвером антивирусной защиты. При этом во входном буфере по нулевому смещению находится байт, значение которого определяет то, какой системный вызов следует использовать (например, для NtCreateFile() это значение равно 0x2713). Все остальное пространство входного буфера используется для хранения указателей на структуры (UNICODE_STRING, OBJECT_ATTRIBUTES и другие), которые следует заполнить и передать в качестве параметров для системного вызова.

Уязвимость, позволяющая выполнить произвольный код с наивысшими привилегиями, содержится в функции, которая непосредственно осуществляет системный вызов. Она заключается в отсутствии проверок рассмотренных выше указателей на параметры системного вызова:

```
int __thiscall sub_288AA(void *this, int InputBuffer, int a3, int a4)
{
      // извлечение параметров для системного вызова из входного буфера
      v18 = *(_DWORD *)(InputBuffer + 56);
      v17 = *(_DWORD *)(InputBuffer + 32);
      v12 = *(_DWORD *)(InputBuffer + 36);
```

```
        v14 = *(_DWORD *)(InputBuffer + 40);
        v15 = *(_DWORD *)(InputBuffer + 44);
        v11 = (HANDLE *)(a3 + 8);
        ObjAttr = *(_DWORD *)(a3 + 0x3C);
        v10 = (int)this;
        StringBuffer = *(_DWORD *)(InputBuffer + 0xC);
        v13 = *(_DWORD *)(InputBuffer + 52);
        UnicodeString = *(_DWORD *)(InputBuffer + 0x44);
        v19 = *(struct _IO_STATUS_BLOCK **)(a3 + 0x40);
        StringLen = *(_DWORD *)(InputBuffer + 0x14);
        v16 = *(_DWORD *)(InputBuffer + 48);
        if (StringBuffer && UnicodeString && ObjAttr && v19 && StringLen)
        {
// заполнение структуры UNICODE_STRING
                *(_DWORD *)(UnicodeString + 4) = StringBuffer;
                *(_WORD *)(UnicodeString + 2) = StringLen;
                *(_WORD *)UnicodeString = StringLen;
// заполнение структуры OBJECT_ATTRIBUTES
                *(_DWORD *)ObjAttr = 24;
                *(_DWORD *)(ObjAttr + 4) = 0;
                *(_DWORD *)(ObjAttr + 12) = 0x240u;
                *(_DWORD *)(ObjAttr + 8) = UnicodeString;
                *(_DWORD *)(ObjAttr + 16) = 0;
                *(_DWORD *)(ObjAttr + 20) = 0;
// вызов оригинального обработчика системного вызова NtCreateFile()
                result = sub_185C2(v10, v11, v12, (OBJECT_ATTRIBUTES
*)ObjAttr, v19, 0, v13, v14, v15, v16, 0, 0, a4);
                if (result < 0)
                *v11 = (HANDLE)-1;
        }
        else
        {
                result = 0xC000000Du;
                *(_DWORD *)(InputBuffer + 4) = 0xC000000Du;
        }
return result;
}
```

Таким образом, путем передачи уязвимому драйверу специальным образом сформированного буфера атакующий может переписать произвольным значением произвольный байт памяти в пространстве ядра.

Стоит отметить, что рассмотренная уязвимость, несмотря на возможность локального выполнения произвольного кода в пространстве ядра, имеет низкую степень опасности. Это связанно с тем, что необходимое для эксплуатации уязвимости устройство \Device\TmComm может быть открыто только пользователем с наивысшими привилегиями.

Можно сделать вывод, что уязвимость представляет исключительно образовательную ценность, и ее использование в реальных условиях является бессмысленным.

Также было проведено тестирование браузера Google Chrome версии 32, в ходе которого была найдена уязвимость и целочисленное переполнение в коде обработки аудиофайлов, однако процесс браузера аварийно завершается прежде, чем появляется возможность эксплуатировать уязвимость. Уязвимость наблюдается в компоненте V8:

```
    (c4ec.c728):Access violation-code c00000005 (first chance) First
chance except are reoirted any exception handling eax=3e700000
ebx=00000006 ecx=3e717101edx=3e000001 esi=215660b4 edi=1c6db928
eip=68eee529 esp=0030edf0 ebp=0030ee00 iopl=0       nv up ei pl zr na pe
nc cs=0023 ss=002b ds=002b es=002b fs=0053 gs =002b  efl=00010206
chrome_child!v8::internal::MarkCompactCollector::MigrateObject+0xb1:68eee5
29 8b500c   nov   edx,dvord ptr [eax+0Ch] ds:002d:3e70000c=????????
```

Кроме того, были обнаружены 10 ошибок разыменования нулевого указателя, которые могут представлять лишь проблему стабильности. И в этом случае выявленные уязвимости реально могут быть использованы только в образовательных целях в качестве иллюстрирующего материала.

## Заключение

Разработанный алгоритм и скрипт, реализующие методику фаззинг-тестирования, в экспериментальном исследовании известных общедоступных программ продемонстрировали свою работоспособность при детектировании уязвимостей. В данной работе основное внимание было уделено решению задачи практической реализации методики фаззинга, поскольку в литературе, посвященной фаззингу, именно в области конкретных программных решений ощущается недостаток предоставляемой информации. Некоторые вопросы, требующие отдельного рассмотрения, вышли за рамки данной работы. К таким вопросам относятся, например, применение технологий распараллеливания алгоритмов, анализ сложности применяемых алгоритмов, проблемы выявления программных закладок и ряд других. В дальнейшем предполагается подробно рассмотреть эти вопросы и опубликовать результаты исследований.

Результаты данной работы представлялись и обсуждались на двух международных научных конференциях [9, 10].

## Литература

1. Саттон М., Грин А., Амини П. Fuzzing: исследование уязвимостей методом грубой силы // Программисту [Электронный ресурс]. 17.09.2018. – URL: http://i.booksgid.com/web/online/42526/ (дата обращения 17.09.2018).

2. База данных эксплойтов от Offensive Security (создателей Kali Linux) // Безопасность [Электронный ресурс]. 16.09.2018. – URL: https://webware.biz/?p=4207/ (дата обращения 16.09.2018).

3. CVE and CCE Statistics Query Page // CVE and CCE [Электронный ресурс]. 18.09.2018. – URL: http://web.nvd.nist.gov/view/vuln/statistics (дата обращения 18.09.2018).

4. Anly K. The Shellcoder's Handbook: Discovering and Exploiting Security Holes // Files [Электронный ресурс]. 17.09.2018. – URL: https://is.gd/my86QP/ (дата обращения 17.09.2018).

5. Krill P. Coverity buys Solidware to boost code analysis // article [Электронный ресурс]. 16.09.2018. – URL: https://www.infoworld.com/article/2642118/application-development/coverity-buys-solidware-to-boost-code-analysis.html/ (дата обращения 16.09.2018).

6. HackTools // Fuzzing [Электронный ресурс]. 16.09.2018. – URL: https://kali.tools/all/?category=fuzzer/ (дата обращения 16.09.2018).

7. Lcamtuf's blog // blogspot [Электронный ресурс]. 18.09.2018. – URL: http://lcamtuf.blogspot.ru/ (дата обращения 18.09.2018).

8. Fuzzing // Fuzzing [Электронный ресурс]. 16.09.2018. – URL: https://www.owasp.org/index.php/Fuzzing/ (дата обращения 16.09.2018).

9. Томилов И. О., Трифанов А. В. Фаззинг. Поиск уязвимостей в программном обеспечении без наличия исходного кода // Интерэкспо ГЕО-Сибирь-2017: XIII Международный научный конгресс. Магистерская научная сессия «Первые шаги в науке»: сборник материалов. Т. 2. (Новосибирск, 17-21 апреля 2017 г.) – Новосибирск: СГУГиТ, 2017. С. 74-80.

10. Томилов И. О., Грицкевич Е. В. Анализ актуальных киберугроз и средств защиты от них // Интерэкспо ГЕО-Сибирь-2018: XIV Международный научный конгресс. Магистерская научная сессия «Первые шаги в науке»: сборник материалов. (Новосибирск, 23-27 апреля 2018 г.) – Новосибирск: СГУГиТ, 2018. С. 99-105.

## References

1. Sutton M., Green A., Amini P. *Fuzzing: issledovaniye uyazvimostey metodom gruboy sily* [Fuzzing: study of vulnerabilities by brute force method]. *Programmistu*, 17 September 2018. Available at: http://i.booksgid.com/web/online/42526/ (accessed 17 September 2018) (in Russian).

2. *Baza dannykh eksploytov ot Offensive Security (sozdateley Kali Linux)* [Database of exploits from Offensive Security (creators of Kali Linux)]. *Bezopasnost'*, 16 September 2018. Available at: https://webware.biz/?p=4207/ (accessed 16 September 2018).

3. CVE and CCE Statistics Query Page. *CVE and CCE*, 18 September 2018. Available at: http://web.nvd.nist.gov/view/vuln/statistics (accessed 18 September 2018).

4. Anly K. The Shellcoder's Handbook: Discovering and Exploiting Security Holes. *Files*, 17 September 2018. Available at: https://is.gd/my86QP/ (accessed 17 September 2018).

5. Krill P. Coverity buys Solidware to boost code analysis. Article, 16 September 2018. Available at: https://www.infoworld.com/article/2642118/application-development/coverity-buys-solidware-to-boost-code-analysis.html/ (accessed 16 September 2018).

6. Hack Tools. *Fuzzing*, 16 September 2018. Available at: https://kali.tools/all/?category=fuzzer/ (accessed 16 September 2018).

7. Lcamtuf's blog. *Blogspot*, 18 September 2018. Available at: http://lcamtuf.blogspot.ru/ (accessed 18 September 2018).

8. Fuzzing. *Fuzzing*, 16 September 2018. Available at: https://www.owasp.org/index.php/Fuzzing/ (accessed 16 September 2018).

9. Tomilov I. O., Trifanov A. V. Fazzing. Poisk uyazvimostey v programmnom obespechenii bez nalichiya iskhodnogo koda [Fuzzing. Search for vulnerabilities in software without source code]. *Interekspo GEO-Sibir'-2017: XIII Mezhdunarodnyy nauchnyy kongress. Magisterskaya nauchnaya sessiya "Pervyye shagi v nauke": sbornik materialov* [XIII International Scientific Congress "Interexpo GEO-Siberia-

2017”. Master's scientific session "The first steps in science"]. Novosibirsk. Siberian state University of geosystems and technologies. 2017, vol. 2, pp. 74-80 (in Russian).

10. Tomilov I. O., Gritskevich E. V. *Analiz aktual'nykh kiberugroz i sredstv zashchity ot nikh* [Analysis of actual cyber threats and means of protection against them]. *Interekspo GEO-Sibir'-2018: XIV Mezhdunarodnyy nauchnyy kongress. Magisterskaya nauchnaya sessiya “Pervyye shagi v nauke”: sbornik materialov* [XIV International Scientific Congress “Interexpo GEO-Siberia-2018”. Master's scientific session "The first steps in science"]. Novosibirsk. Siberian state University of geosystems and technologies. 2018, pp. 99-105 (in Russian).

**Статья поступила 25 сентября 2018 г.**

## Информация об авторах

*Томилов Илья Олегович* – соискатель ученой степени кандидата технических наук. Инженер отдела кибербезопасности. Сбербанк России. Область научных интересов: информационная безопасность; киберугрозы; средства тестирования безопасности; поиск уязвимостей в приложениях; технологии автоматизированного тестирования; тестирование безопасности; аудит; фаззинг; оценка защищённости информационных систем; тестирование на проникновение. E–mail: tio28@yandex.ru

*Карманов Игорь Николаевич* – кандидат технических наук, доцент. Заведующий кафедрой информационной безопасности. Сибирский государственный университет геосистем и технологий. Область научных интересов: информационная безопасность, физическая оптика, физика лазеров. E–mail: i.n.karmanov@ssga.ru

*Звягинцева Полина Александровна* – соискатель ученой степени кандидата технических наук. Старший преподаватель кафедры информационной безопасности. Сибирский государственный университет геосистем и технологий. Область научных интересов: информационная безопасность, оптотехника. E–mail: polinasgugit@mail.ru

*Грицкевич Евгений Владимирович* – кандидат технических наук, доцент. Доцент кафедры информационной безопасности. Сибирский государственный университет геосистем и технологий. Область научных интересов: информационная безопасность систем и технологий оптотехники. E–mail: gricew@mail.ru

Адрес: 630108, Россия, г. Новосибирск, ул. Плахотного, д. 10.

---

# Development of fuzzing application technique for software vulnerabilities analysis

I. O. Tomilov, I. N. Karmanov, P. A. Zvyagintseva, E. V. Gritskevich

**Relevance**: currently, information resources become the main target of attacks by intruders. The attacking party actively exploits the most common system vulnerabilities: memory leaks, encryption weaknesses, buffer overflow, incomplete input data checking, etc. The attacked party, in turn, tries to find system vulnerabilities by developing and improving barriers that defend the protected information object from unauthorized access. To protect software and information resources, standard methods are often used to ensure

the information security of a system and its software. The most common security approach is to periodically update the system and applications. The update closes vulnerabilities and fixes errors that were made in programs at the time of their creation. However, it does not take into account the relevance of threats and the real demand for a particular update. Such a concept does not provide an integrated approach to security. **Purpose**: to develop a system of fuzzing software testing for the presence of vulnerabilities in it with inclusion of a so-called intelligent agent in the system. The **object of the study** is the process of detecting software vulnerabilities. The **subject of research** is the fuzzing technology used for search of vulnerabilities. **Used methods**: the solution to the problem is the dynamic analysis of the process of program code execution. The advantage of this approach is that with this testing, false positives are almost completely absent, in contrast to static analyzers. **Novelty**: elements of practical novelty are the definition of requirements for a testing system and the development of an algorithm and methodology for fuzzing testing. **Result**: the developed algorithm and script for fuzz testing showed its efficiency in finding vulnerabilities in the experimental study of known public programs. **Practical significance**: the detection of even a single vulnerability in the short term in automatic mode can be considered as the evidence of reliability of the technology which allows effective blocking of possible attacks by intruders.

***Key words**: fuzzing, software vulnerabilities, software testing, cyber threats, software audit, intelligent agent.*

## Information about Authors

*Ilia Olegovich Tomilov* – Doctoral Student. Engineer of the Cyber Security Department. Public Joint Stock Company "Sberbank of Russia". Fields of research: information security; cyberthreats; security testing tools; search for vulnerabilities in applications; technology of automated testing; security testing; audit; fuzzing; the evaluation of the security of information systems; penetration testing. E–mail: tio28@yandex.ru

*Igor Nikolaevich Karmanov* – Ph.D. of Engineering Sciences, Associate Professor. Head of Department of Information Security. Siberian state University of geosystems and technologies. Fields of research: information security, physical optics, laser physics. E–mail: i.n.karmanov@ssga.ru

*Polina Alexandrovna Zviaginceva* – Doctoral Student. Senior Lecturer of Department of Information Security. Siberian state University of geosystems and technologies. Fields of research: information security, optoelectronics. E–mail: polinasgugit@mail.ru

*Evgenij Vladimirovich Gritskevich* – Ph.D. of Engineering Sciences, Associate Professor. Ass. Prof. of Department of Information Security. Siberian state University of geosystems and technologies. Field of research: information security of systems and technologies of optoelectronics. E–mail: gricew@mail.ru

Address: Russia, 630108, Novosibirsk, Plakhotnogo, 10.