Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах использующих сетевые сокеты[1]

Исаев И.К., Сидоров Д.В., Герасимов А.Ю., Ермаков М.К.,
iisaev@ispras.ru, sidorov@ispras.ru, agerasimov@ispras.ru, mermakov@ispras.ru

Аннотация. В данной статье рассматривается модификация и применение инструмента Avalanche для проведения динамического анализа и тестирования приложений, получающих входные данные через сокеты. Вводится концепция замены получаемых данных, описывается реализация этой концепции при помощи средств Valgrind. Разбирается перехват и обработка системных вызовов, используемых при работе с сокетами. Рассматривается применение модифицированной версии инструмента для анализа сетевых приложений с открытым исходным кодом, перечисляются обнаруженные во время анализа дефекты.

Ключевые слова: динамический анализ; обнаружение ошибок; тестирование.

1. Введение.

Известно, что проблемам надежности программ, работающих с данными, получаемыми через сеть, уделяется традиционно повышенное внимание. Наличие уязвимости в такой программе может отразиться на всех её пользователях, поскольку данные, вызывающие возникновение ошибки, могут быть быстро переданы по сети большому числу пользователей. Значительные усилия тратятся на борьбу с уязвимостями безопасности в Web-приложениях.

Инструмент Avalanche в своем исходном виде использует динамический анализ для создания входных данных, демонстрирующих ошибки в анализируемой программе. При этом в качестве источника входных данных для программы поддерживается лишь единственный файл, с которым работает программа. Расширение списка возможных источников входных данных для программы так, чтобы он включал чтение данных из сокета, - основная цель данной работы. При этом результат анализа, как и в немодифицированной

[1] Работа поддержана грантом РФФИ №11-07-00466-а и ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы» ГК №07.514.11.4040

версии инструмента Avalanche, должен являться не только список обнаруженных ошибок, но и набор входных данных для программы, приводящих к её возникновению.

Инструмент Avalanche изначально представлен в работе [1]. Напомним кратко его структуру и основные свойства. Для описания Avalanche вводится понятие символических или помеченных (tainted) данных - данных, полученных программой из внешнего источника (стандартный поток ввода, файлы, переменные окружения и т. д.)

Avalanche состоит из 4 основных компонент (см. Рисунок 1): двух модулей расширения (плагинов) Valgrind - Tracegrind и Covgrind, инструмента проверки выполнимости ограничений STP и управляющего модуля. Tracegrind динамически отслеживает поток помеченных данных в анализируемой программе и собирает условия для обхода её непройденных частей и для срабатывания опасных операций. Управляющий модуль передаёт собранные ограничения STP для проверки их выполнимости. Если какие-то из условий выполнимы, то STP определяет те значения всех входящих в условия переменных (в том числе и значения байтов входного файла), которые обращают условие в истину.

- Если какое-то из ограничений для выполнения опасной операции выполнимо, управляющий модуль запускает программу ещё раз (на этот раз безо всякой инструментации) с соответствующим входным файлом для того, чтобы подтвердить обнаруженную ошибку.
- Выполнимые ограничений для обхода ранее не пройденных частей программы определяют набор входных данных для новых запусков программы. Таким образом, после проведения одной итерации анализа STP автоматически вычисляет набор новых входных данных для последующих итераций. Из этого набора необходимо выбрать определённое значение для запуска следующей итерации. При этом в первую очередь должны обрабатываться входные данные на которых наиболее вероятно возникновение ошибки. Для решения этой задачи используется эвристическая метрика - количество ранее не обойденных базовых блоков в программе. Для измерения значения эвристики используется компонент Covgrind, в функции которого входит также фиксация возможных ошибок выполнения. Covgrind - гораздо более легковесный модуль, нежели Tracegrind, поэтому возможно сравнительно быстро измерить значения звристики для всех полученных ранее входных файлов и выбрать входной файла с наибольшим её значением.
- Covgrind может обнаруживать только те ошибки, которые приводят к аварийному завершению программы. Для того, чтобы обнаруживать иные типы ошибок, может быть использован один из наиболее популярных инструментов Valgrind – Memcheck.

Более подробное описание исходной версии Avalanche можно найти в статье [1]. Данная статья является её непосредственным логическим продолжением. Здесь используется та же терминология, что и в [1], опускается существенная часть вопросов, ранее рассмотренных в [1] и т. п. Работа имеет следующую структуру. В разделе 2 вводится концепция замены получаемых входных данных, описывается общая схема работы. В разделе 3 описывается специальный формат файла, который Avalanche использует для запуска анализируемого приложения с различными входными данными. В разделе 4 описывается перехват системных вызовов, используемых при взаимодействии клиента с сервером. В разделе 5 приводится алгоритм замены получаемых входных данных. В шестом разделе перечисляются изменения в компонентах Tracegrind и Covgrind, необходимые для поддержки сокетов. В разделе 7 рассматриваются результаты работы модифицированной версии Avalanche, перечисляются обнаруженные дефекты. Раздел 8 описывает имеющиеся ограничения предложенного метода, а также намечает направления для дальнейшего исследования.

2. Организация анализа

2.1. TCP и UDP сокеты

Большинство сетевых приложений используют архитектуру клиент-сервер. Для организации взаимодействия между удаленными процессами на транспортном уровне используются протоколы TCP (ориентированный на установку соединения) и UDP (ориентированный на отправку сообщений). Типичная последовательность системных вызовов на стороне клиента при использовании протокола TCP имеет вид:

- socket
- connect
- read/write или send/recv

Сервер же будет использовать иную последовательность:

- socket
- bind
- listen
- accept
- read/write или send/recv

В случае протокола UDP последовательности системных вызовов также будут различными. Поскольку Avalanche использует возможности Valgrind[2] для динамической инструментации и перехвата системных вызовов, то различные последовательности системных вызовов требуют, вообще говоря, различных

моделей работы. В данной работе рассматривается анализ клиентских приложений, использующих сокеты, ориентированные на установку соединения (TCP-сокеты).

2.2. Взаимодействие с сервером

Если в случае простого чтения из файла для запуска программы на новых входных данных достаточно лишь изменить содержимое этого файла, то в случае клиент-серверного взаимодействия запуск программы на новых входных данных не так прост. Очевидно, что сокеты не могут быть источником потенциально опасных данных в случае изолированной работы анализируемого приложения-клиента. С точки зрения логики работы сетевого приложения, клиент составляет лишь часть единой программы – связки клиент-сервер. Поэтому решение вопроса об организации взаимодействия анализируемого приложения с сервером является одним из ключевых для успешного проведения анализа.

Допустим, что во время анализа приложения-клиента доступно соответствующее приложение-сервер. Таким образом, клиент может осуществлять обмен данными с сервером в обычном режиме. Тогда для того, чтобы заставить приложение-клиент работать с измененными входными данными, достаточно подменить данные приходящие от сервера, на данные, полученные в результате проверки выполнимости условий, собранных при предыдущих запусках программы. Такую подмену нужно производить непосредственно в местах появления этих данных в программе, т. е. сразу после выполнения вызовов read или recv, читающих данные из сокетов, соответствующих соединению с сервером.

С использованием такого подхода общая схема работы инструмента Avalanche претерпевает следующие изменения. При первом запуске компонента Tracegrind, помимо обычных файлов с запросами для проверки выполнимости условий, формируется файл специального формата, содержащий все данные, полученные анализируемым приложением от сервера. После проверки выполнимости собранных условий содержимое этого файла изменяется управляющим модулем: в него поочередно подставляются те значения, которые делают эти условия выполнимыми (проверка выполнимости и подбор соответствующих значений, как и при работе с файлами, осуществляются компонентом STP [3]). При всех последующих запусках компонентов Covgrind и Tracegrind они заменяют приходящие от сервера данные в соответствии с содержимым модифицированных файлов с данными. Кроме того, если во время работы компонента Tracegrind приложение получает от сервера порцию данных большего размера, чем есть в файле с данными для замены, то “остаток” полученных от сервера данных приписывается к файлу с данными для замены в неизменном виде. Далее рассмотрим общую схему более детально.

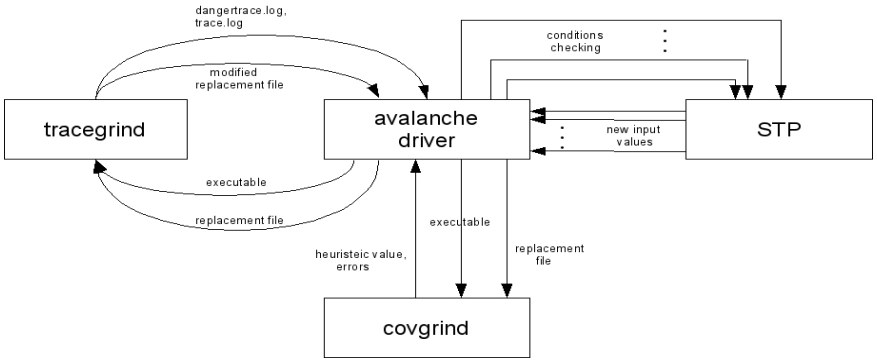

Рис. 1. Общая схема

3. Формат файла с данными для замены

Управляющий модуль по результатам проверки выполнимости формулы компонентом STP формирует файл, содержащий такие данные, что при их получении клиентом со стороны сервера выполнение должно пойти по новому пути. Этот файл имеет следующий вид:

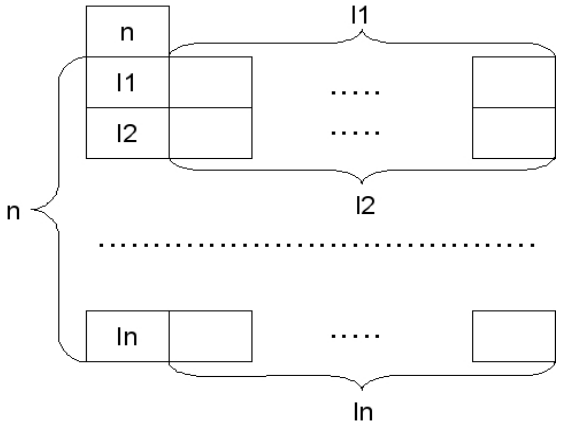

Рис. 2. Формат файла с данными для замены.

Первые четыре байта содержат целое число n – количество различных соединений клиента и сервера (оно равно количеству вызовов connect со

стороны клиента в анализируемой части трассы клиента при предыдущем запуске). Далее следует n групп. В i-ой группе первые четыре байта – это длина группы li (т. е. число байт, прочитанных клиентом по i-ому соединению), а следующие li байт – это собственно полученные клиентом данные.

В таком же формате сохраняются и данные, приводящие к возникновению ошибки в исследуемом приложении.

4. Перехват системных вызовов

Поток потенциально опасных данных в анализируемом приложении отслеживается аналогично тому, как это делалось в программах, работающих с обычными файлами. Однако есть одно существенное различие – иной источник потенциально опасных данных. Для того, чтобы зафиксировать факт получения данных через сокет, приходится перехватывать следующие системные вызовы, ответственные за работу с сокетами:

- connect
- read
- recv
- close

Эти системные вызовы необходимо перехватывать как в компоненте Tracegrind, так и в компоненте Covgrind, поскольку, хоть Covgrind и не отслеживает поток потенциально опасных данных в программе, но оба компонента должны осуществлять замену приходящих от сервера данных при их получении клиентом.

В общем случае, клиент может устанавливать за время работы несколько соединений с сервером. Каждое установленное соединение идентифицируется при помощи сквозной нумерации – все соединения нумеруются согласно порядку их установления, начиная с нуля. Оба компонента поддерживают во время работы внутреннюю структуру данных, позволяющую устанавливать соответствие между номерами дескрипторов сокетов и номером соединения, а также подсчитывать объем данных, полученных по каждому из соединений.

Если в случае чтения данных из файла для его идентификации используется имя, то сетевое соединения идентифицируется парой IP-адрес плюс номер порта. При перехвате системного вызова connect анализируется второй параметр вызова – адрес сервера, с которым устанавливается соединение. IP-адрес и номер порта, по которому сервер ждёт подключения от клиента, должны быть переданы Avalanche в качестве параметров запуска (которые управляющий модуль затем передаёт Tracegrind при каждом его запуске). Если IP-адрес и номер порта, содержащиеся во втором параметре успешно завершившегося вызова connect совпадают с IP-адресом и номером порта,

указанными в параметрах запуска, то такой вызов connect устанавливает соединение клиента с сервером, и все данные переданные по этому сокету помечаются как потенциально опасные. При этом запоминается, что первый параметр вызова connect (дескриптор сокета) соответствует очередному соединению. Счетчик полученных данных для нового соединения устанавливается равным нулю.

5. Замена данных при работе компонентов Tracegrind и Covgrind

При перехвате системных вызовов read или recv анализируется их первый параметр – дескриптор, по которому осуществляется чтение данных. Если этот дескриптор соответствует какому-то из установленных и не закрытых соединений, то такой вызов является источником потенциально опасных данных в программе.

При перехвате этих вызовов и осуществляется непосредственная замена данных. При этом если количество байт уже полученных по тому соединению, из которого осуществляется чтение, не превосходит количества байтов в соответствующем файле с данными для замены, то прочитанные данные заменяются на данные из файла. В противном случае замены не происходит, и анализируемое клиентское приложение получает от сервера именно те данные, которые и были переданы.

Поясним данное правило на примере. Пусть перехвачен вызов read, читающий ri байт из сокета, соответствующего i-ому соединению. Пусть ранее из этого соединения было прочитано bi байт, а длина соответствующей порции данных в файле с данными для замены равна li. Тогда возможны следующие варианты:

a) bi + ri <= li

В этом случае все ri прочитанных байт заменяются на байты с номерами с bi + 1 по bi + ri в i-ой группе байтов из файла с данными для замены.

b) bi < li, bi + ri > li

Тогда первые li - bi из прочитанных ri байт заменяются на байты с номерами с bi + 1 по li в i-ой группе байтов из файла с данными для замены, а последующие ri - (li - bi) байтов передаются программе без изменений. Кроме того, в случае работы компонента Tracegrind, эти ri - (li - bi) байтов дописываются в конец i-ой группы.

c) bi >= li

Тогда все прочитанные данные передаются программе без изменений и, кроме того, в случае работы компонента Tracegrind, эти данные дописываются в конец i-ой группы.

После этого счетчик прочитанных данных увеличивается на ri: bi += ri.

Системный вызов recv обрабатывается аналогично за одним исключением. Если в четвёртом параметре вызова recv установлен флаг MSG_PEEK, то это означает, что прочитанные данные не удаляются из очереди полученных данных, и последующие обращения к вызовам read или recv прочитают те же данные. В таком случае счетчик прочитанных данных после завершения обработки увеличивать не надо.

6. Изменения в компонентах Tracegrind и Covgrind

6.1. Моделирование сокетов

Сокеты представляются аналогично тому, как это делается с входным файлом, адресным пространством программы и набором регистров в обычной версии Avalanche. Каждое установленное соединение представляется в виде массива с 32-битными индексами. При перехвате каждого успешного вызова connect, устанавливающего соединение с сервером, в файлы с STP-декларациями добавляется декларация вида

```
socket_2 : ARRAY BITVECTOR(32) OF BITVECTOR(8);
```

Здесь 2 – это порядковый номер установленного соединения (начиная с 0).

6.2. Декларации для системных вызовов

При перехвате системных вызовов read и recv создаётся набор деклараций, отражающих факты равенства значений, прочитанных из сокета, содержимому ячеек памяти, по которым осуществлялась запись:

```
memory_1 : ARRAY BITVECTOR(32) OF BITVECTOR(8) =
memory_0 WITH [0hex08090218] := socket_0[0hex00000000];
```

Данная декларация моделирует чтение первого байта, полученного по первому соединению, установленному анализируемым приложением, в ячейку памяти с адресом 0x08090218. Аналогично чтению из файла, каждый из вызовов read и recv приводит к появлению цепочки таких деклараций – по числу прочитанных байт.

6.3. Тайм-аут для предотвращения блокировки анализа

Как уже упоминалось, для поддержки запуска программы на различных значениях входных данных, компонент Covgrind вынужден перехватывать системные вызовы, работающие с сокетами и осуществлять подмену данных. Кроме того, изменилась семантика таймера, используемого компонентом Covgrind. Если в случае анализа приложения, осуществляющего чтение из файла, значение устанавливаемого при старте приложения таймера означало максимальное допустимое время работы приложения, и его истечение интерпретировалось как показатель наличия в программе бесконечного цикла

(т. е. ошибки), то в случае анализа сетевых приложений продолжительную работу клиента можно трактовать как ошибку с гораздо меньшей степенью уверенности. Длительная работа клиента может быть связана, например, со спецификой конкретного сетевого приложения (например, проигрывание потокового аудио) или с ожиданием отклика сервера. Поэтому при анализе сетевых клиентских приложений таймер используется для борьбы с блокировкой анализа – Covgrind измеряет покрытие анализируемой программы и фиксирует наличие ошибок только за отведённое таймером время. По истечении таймера работа Covgrind прерывается, но это не считается признаком наличия ошибки в анализируемой программе.

6.4. Воспроизведение обнаруженного дефекта

Помимо этого, Covgrind используется для воспроизведения найденной ошибки. Если в случае простого чтения данных из файла для воспроизведения ранее обнаруженной ошибки достаточно запустить программу с предварительно сохранённым файлом, то в случае сетевого приложения продемонстрировать уязвимость сложнее, даже имея соответствующие входные данные. Можно было бы для каждой обнаруженной ошибки создать сервер, отсылающий соответствующие данные клиенту, но это, в общем случае, требует хорошего знания протокола, по которому сервер общается с клиентом, и не позволяет автоматически (т. е. без вмешательства эксперта) воспроизводить ошибочную ситуацию. Однако, можно использовать всё ту же технику замены приходящих от сервера данных на данные, сгенерированные Avalanche и демонстрирующие уязвимость программы. Таким образом, для воспроизведения ошибки достаточно отдельно (вне окружения Avalanche) запустить Covgrind с ранее сохранённым файлом в соответствующем формате (см пункт 2).

7. Результаты работы Avalanche на реальных приложениях, получающих данные через сокеты

Эффективность поиска ошибок при помощи Avalanche была исследована на большом числе проектов с открытым исходным кодом. В данном разделе будут рассмотрены результаты работы на следующих 6 проектах:

- fetchmail-6.3.14
  Fetchmail – утилита, используемая для сбора почты с удалённых POP3, IMAP, ETRN или ODMR почтовых серверов и доставки локальным пользователям.
- mencoder (версия из репозитория проекта)
  MEncoder – инструмент кодирования/декодирования и фильтрации данных в аудио и видео форматах. Является частью проекта MPlayer, может использоваться для конвертации файлов во всех форматах, поддерживаемых MPlayer. Поддерживает работу с потоковым аудио, получаемым от удаленного сервера.

- nmap-5.21
  Nmap — утилита, предназначенная для настраиваемого сканирования IP-сетей и определения состояния объектов сканируемой сети (портов и соответствующих им служб). Поддерживает в том числе и TCP-сканирование.
- wget-1.12
  Wget – неинтерактивная консольная программа для загрузки файлов по сети. Поддерживает протоколы HTTP, FTP и HTTPS.
- ogg123 (vorbis-tools-1.2.0)
  Ogg123 – проигрыватель для воспроизведения медиаданных в форматах Ogg Vorbis, Ogg Speex, Ogg FLAC. Как и MEncoder, поддерживает работу с потоком данных, получаемых от удаленного сервера.

В соответствии с концепцией замены приходящих от сервера данных все тестируемые приложения во время анализа взаимодействовали с “ожидаемым” приложением сервером (т. е. с сервером, поддерживающим тот же протокол, что и анализируемое приложение). Для анализа mencoder и ogg123 использовался сервер потокового аудио Icecast. Fetchmail работал с почтовым сервером Dovecot. Nmap осуществлял TCP-сканирование порта, на котором также работал почтовый сервер Dovecot. Wget загружал файл с HTTP-сервера nginx.

При анализе fetchmail, ogg123 и wget устанавливался шестисекундный тайм-аут для работы компонента Covgrind. Глубина просмотра во всех случаях указывалась равной 100 условиям. На работу Avalanche отводилось 3600 секунд (час), затем анализ принудительно завершался. Результаты работы приведены в таблице 1.

Перечислим обнаруженные дефекты:

- mencoder
  Дефект связан с разыменованием нулевого указателя. Возникал при ошибке разбора пакета, полученного от сервера. Был исправлен разработчиком в течение суток после момента сообщения.
- wget
  Аналогично предыдущему случаю, разыменование нулевого указателя при разборе полученного пакета. Разработчик сообщил, что дефект уже был ранее исправлен в текущей разрабатываемой версии приложения.
- nmap, ogg123, fetchmail
  Дефектов в результате анализа не обнаружено.

По-видимому, все обнаруженные дефекты реализуются в ситуации, когда клиент получает от сервера “неожиданные” данные, то есть данные, получение которых не предусмотрено протоколом взаимодействия. Таким образом, при полностью корректной работе сервера подобные ошибки в приложении-клиенте никак себя проявлять не будут. Однако в случае, если при реализации протокола на стороне сервера допущена ошибка, или в качестве сервера потенциально может выступать «ненадежное» приложение

(т. е. приложение, намерено пытающееся спровоцировать ошибку на стороне клиента), такие ошибки обычно приводят к аварийному завершению приложения-клиента, и могут быть классифицированы как серьезные уязвимости.

	fetchmail	mencoder	nmap	wget	ogg123
общее число тестов	396	163	163	613	36
начальное покрытие	8199	7751	17131	6833	9154
Прирост покрытия	41 (0.5%)	226 (3%)	380 (2.2%)	39 (0.5%)	103 (1.1%)
точность предсказания	79%	100%	100%	93%	100%
время tracegrind	35%	0.3%	38.4%	10.5%	1%
время STP	8%	91%	<1%	1%	94%
время covgrind	57%	8.7%	61%	88.5%	5%
опасные операции	0	0	0	0	0
число дефектов	0	1	0	1	0
тестовые данные	0	1	0	9	0
tmin	-	98	-	95	-
tmax	-	98	-	95	-

Таблица 1.Результаты и статистика работы Avalanche при анализе сетевых клиентских приложений

8. Ограничения и направления дальнейшего развития

Стоит ещё раз подчеркнуть, что анализ приложения требует наличия сервера, работающего по понятному приложению протоколу. Изолированный анализ приложения требовал бы восстановления протокола лишь по его клиентской части, что представляется крайне затруднительным. При работе над поддержкой сокетов в Avalanche была первоначально предпринята попытка

организовать анализ таким образом, чтобы серверная часть приложения эмулировалась управляющим модулем инструмента Avalanche. При этом подход к эмуляции сервера состоял в следующем: при достижении анализируемым клиентским приложением (во время запуска с инструментирующим модулем Tracegrind или Covgrind) одного из системных вызовов для работы с сокетами (connect, select, poll, read, recv), Tracegrind или Covgrind сообщают об этом вызове управляющему модулю, и тот выполняет ответные действия по обработке соответствующего вызова. В частности, в случае выполнения вызова connect управляющий модуль должен принять новое соединения (выполнить системный вызов accept), а в случае вызовов read или recv записать ранее определённый при помощи компонента STP порцию данных (см. Рисунок 3).

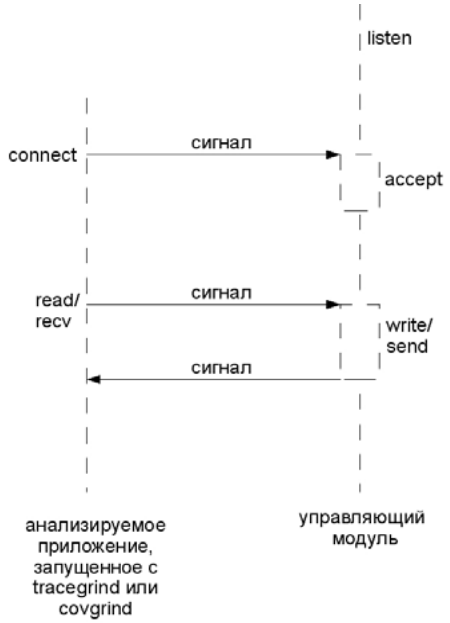

Рис. 3. Схема работы с эмуляцией сервера

Это решало бы проблему запуска программы с новыми входными данными. Однако попытка реализации данного подхода привела к обнаружению ряда сложностей:

1) Необходимость синхронизации между управляющим модулем и анализируемым при помощи Tracegrind или Covgrind приложением. Например, для обработки системного вызова read инструментирующий модуль должен каким-то образом сообщить об этом вызове управляющему модулю и приостановить работу программы, а управляющий модуль

должен, получив соответствующее сообщение, записать необходимую порцию данных, и затем выслать инструментирующему модулю подтверждение о завершении обработки, чтобы тот мог продолжить свое выполнение. В сочетании со сложностью межпроцессного взаимодействия в рамках фреймворка Valgrind это приводило к существенному усложнению компонентов Tracegrind и Covgrind.

2) Проблема обратной инженерии протокола взаимодействия клиента и сервера. Поскольку в общем случае про анализируемое приложение заранее не известно ничего, то возникает необходимость учёта большого числа различных вариантов при взаимодействии клиента и сервера – использование неблокирующих сокетов, установка ненулевых тайм-аутов в системных вызовах poll и select и т. д. Приходится либо вводить жесткие ограничения на анализируемые приложения (вида “анализируемое приложение должно использовать только блокирующие сокеты” и т. п.), либо пытаться организовать эмуляцию сервера управляющим модулем с поддержкой всех возможных вариантов, что крайне трудоёмко, если вообще возможно.

Поэтому поиск дефектов с эмуляцией сервера управляющим модулем успешно работал только на простых, искусственно созданных примерах уязвимых приложений. В итоге успешного анализа удалось добиться, только используя замену данных, приходящих от независимо работающего сервера.

Кроме того, как уже было сказано, на данный момент поддержка сокетов в качестве источника потенциально опасных данный реализована в инструменте Avalanche только для клиентских приложений и только для TCP-сокетов. Очевидно, встает вопрос о возможности поддержи UDP-сокетов, а также о возможности анализа серверной части приложений и о применимости концепции замены данных к этим случаям. Исследование этих вопросов привело к следующим выводам.

По-видимому, поддержка UDP-сокетов представляет чисто техническую задачу. Напротив, поддержка возможности анализа серверных приложений представляется более сложной задачей. Основные причины этого таковы:

- Приложение-сервер часто представляет собой несколько параллельно работающих процессов.
  Наличие нескольких процессов, одновременно работающих с потенциально опасными данными, усложняет сбор трассы и генерацию запросов для получения новой порции тестовых данных. Придётся собирать отдельные трассы для каждого из таких процессов, что усложнит как компонент Tracegrind, так и управляющий модуль, осуществляющий разбор собранных трасс.
- Необходимость многократного запуска приложения-клиента.
  Как уже было сказано, для замены приходящих данных во время анализа приложения-клиента, необходим работающий сервер. Для этого достаточно запустить такой сервер единственный раз непосредственно

перед началом анализа. При этом многократные запуски клиента по ходу анализа не влияют (по крайней мере, существенно) на работоспособность сервера.

Но многократные запуски и завершения сервера (во время его анализа) в общем случае потребуют и многократного запуска вспомогательного приложения-клиента, поскольку при неработающем сервере многие клиенты прекращают свою работу. Это также усложнит управляющий модуль.

9. Схожие работы

Для автоматического поиска известных уязвимостей могут применяться инструменты – сканеры уязвимостей (Nikto[4], Nessus[5]). Такие инструменты прежде всего сканируют порты и определяют использующие их сервисы. Затем проводится проверка обнаруженных сервисов при помощи ранее составленной базы известных уязвимостей. Такие инструменты весьма полезны при проверке безопасности тестируемых приложений. Однако, в отличие от Avalanche, с их помощью нельзя обнаружить ранее не известную ошибку. По сути, эти инструменты осуществляют тестирование, но не поиск непосредственных ошибок.

Как статический, так и динамический анализ широко применяется для поиска уязвимостей безопасности. В работе [6] используется динамический анализ для тестирования Web-приложений, написанных на языке Python. При этом так же, как и в Avalanche, собирается трасса в виде набора инструкций байт кода. Однако, отличие работы [6] заключается в том, что динамический анализ используется не для расширения покрытия тестируемого приложения, а для более точного составления тестовых данных (например, если какой-то из параметров HTTP-запроса используется только при составлении запроса к базе данных, то это признак того, что данный параметр бесполезно тестировать на внедрения скрипта и т. п.). Кроме того, основное внимание уделяется уязвимостям безопасности, что ставит Avalanche и работу [6] в несколько различные категории.

В работе [7] используется инструментация байт кода Java для предотвращения уязвимостей безопасности в Java Web приложениях. При этом список методов, которые могут быть источниками потенциально опасных данных, является одним из параметров конфигурации, предоставляемых пользователем. Аналогично предоставляется список методов, использование потенциально опасных данных в качестве параметров которых приводит к возникновению ошибки. Инструментирующий код отслеживает распространение потенциально опасных данных в анализируемой программе и вызывает исключение в том случае, если программа пытается использовать потенциально опасные данные в качестве параметров одного из методов списка. Этот подход, хоть и схож в плане инструментации и отслеживанию распространения потенциально опасных данных в программе, не

осуществляет целенаправленный поиск ошибок в программе, а лишь защищает от их возможных последствий.

10. Заключение

Мы рассмотрели поддержку получения потенциально опасных данных через сокеты для анализа сетевых приложений при помощи Avalanche. Основной идеей, реализация которой позволила этого добиться, является концепция замены получаемых приложением “нормальных” входных данных на специально подобранные значения. Несмотря на то, что поддержка анализа реализована пока только для клиентских приложений, есть основания рассчитывать на то, что концепция замены данных является универсальной, т. е. пригодной как для анализа клиентских, так и для анализа серверных приложений. Если удастся её реализовать для анализа серверных приложений, это, вероятно, позволит обнаруживать ещё более ценные ошибки, поскольку, в общем случае, ошибки в программе-сервере являются более критичными, чем ошибки в программе-клиенте.

Список литературы

[1] И. Исаев, Д. Сидоров. "Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах". Программирование, №4 2010.

[2] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In PLDI, 2007.

[3] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In CAV 2007, LNCS 4590, pages 519–531.

[4] Nikto Web Scanner. http://cirt.net/nikto2

[5] Nessus, The Network Vulnerability Scanner. http://www.nessus.org/nessus/

[6] D. Kozlov, A. Petukhov, "Detecting Security Vulnerabilities in Web Applications Using Dynamic Analysis with Penetration Testing". OWASP Application Security Conference 19-22 May 2008, Ghent, Belgium, 2008.

[7] V. Haldar, D. Chandra, M. Franz. "Dynamic Taint Propagation for Java". In: Proceedings of the 21st Annual Computer Security Applications Conference (2005).