

Практическое задание 6

Фаззинг-тестирование

Задачи работы:

1. Необходимо подготовить виртуальную машину с ОС Linux
2. Запустить VM и установить в нее Docker
3. Выполнить команду ``docker pull aflplusplus/aflplusplus``
4. Выбрать фаззинг-цель для тестирования (простое приложение на `C`) и запустить фаззинг-тестирование исследуемой программы.
5. Подготовить отчет со скриншотами и командами, выполненными внутри контейнера.

Фаззинг (или **fuzzing**) — это метод тестирования программного обеспечения, который используется для обнаружения уязвимостей и ошибок в приложениях. Этот подход включает в себя автоматическую генерацию и отправку случайных или некорректных данных (входных данных) в программу с целью выявления сбоев, исключений или других нежелательных реакций.

-y: Этот флаг автоматически отвечает "да" на все запросы, которые могут возникнуть во время установки. Это позволяет избежать необходимости вручную подтверждать установку, что удобно для автоматизации процесса.

```
pavel@debian2ver1210: ~
root@debian2ver1210:/home/pavel# sudo apt install -y docker.io
Чтение списков пакетов... Готово
Построение дерева зависимостей... Готово
Чтение информации о состоянии... Готово
Следующий пакет устанавливался автоматически и больше не требуется:
  linux-image-6.1.0-32-amd64
Для его удаления используйте «sudo apt autoremove».
Будут установлены следующие дополнительные пакеты:
  binutils binutils-common binutils-x86-64-linux-gnu cgroupfs-mount containerd
  criu git git-man iptables libbinutils libctf-nobfd0 libctf0 liberror-perl
  libgprofng0 libintl-perl libintl-xs-perl libip6tc2 libmodule-find-perl
  libnet1 libproc-processtable-perl libsort-naturally-perl
  libterm-readkey-perl needrestart patch python3-protobuf runc tini
Предлагаемые пакеты:
  binutils-doc containernetworking-plugins docker-doc aufs-tools btrfs-progs
  debootstrap rinse rootlesskit xfsprogs zfs-fuse | zfsutils-linux
  git-daemon-run | git-daemon-sysvinit git-doc git-email git-gui gitk gitweb
  git-cvs git-mediawiki git-svn firewallld needrestart-session | libnotify-bin
  iucode-tool ed diffutils-doc
Следующие НОВЫЕ пакеты будут установлены:
  binutils binutils-common binutils-x86-64-linux-gnu cgroupfs-mount containerd
  criu docker.io git git-man iptables libbinutils libctf-nobfd0 libctf0
  liberror-perl libgprofng0 libintl-perl libintl-xs-perl libip6tc2
  libmodule-find-perl libnet1 libproc-processtable-perl libsort-naturally-perl
```

--now: Этот флаг указывает, что служба должна быть запущена немедленно, а не только включена для автоматического запуска при следующей загрузке. Это означает, что команда не только активирует службу, но и запускает ее сразу.

```
root@debian2ver1210:/home/pavel# sudo systemctl enable --now docker
Synchronizing state of docker.service with SysV service script with /lib/systemd
/systemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install enable docker
root@debian2ver1210:/home/pavel#
```

-a: Этот флаг означает "append" (добавить). Он используется для добавления пользователя в указанную группу, не удаляя его из других групп, к которым он уже принадлежит. Без этого флага пользователь был бы удален из всех групп, кроме указанной.

-G: Этот флаг указывает, что следующая часть команды будет содержать список групп, в которые нужно добавить пользователя. В данном случае это группа docker.

```
root@debian2ver1210:/home/pavel# sudo usermod -aG docker $USER
root@debian2ver1210:/home/pavel#
```

newgrp docker # Активация изменений без перезагрузки

```
Executing: /lib/systemd/systemd-sysv-install enable docker
root@debian2ver1210:/home/pavel# sudo usermod -aG docker $USER
root@debian2ver1210:/home/pavel# newgrp docker
root@debian2ver1210:/home/pavel# docker --version
Docker version 20.10.24+dfsg1, build 297e128
root@debian2ver1210:/home/pavel#
```

Команда `dockerpullaflplusplus/aflplusplus` используется для загрузки Docker-образа из DockerHub, который содержит инструмент для фаззинга под названием AFL++.

```
root@debian2ver1210:/home/pavel# docker pull aflplusplus/aflplusplus
Using default tag: latest
latest: Pulling from aflplusplus/aflplusplus
215ed5a63843: Pull complete
e1de6e5df66b: Pull complete
9cb83937b100: Pull complete
854406e96d76: Pull complete
463787f8a86d: Pull complete
f0e1e1e8bcdb: Pull complete
db46af696403: Pull complete
d81a75ae0a3b: Pull complete
b1f4f0a8fde2: Pull complete
be9e7a34634c: Pull complete
2a5127bf37a0: Pull complete
0ba13a69e764: Pull complete
Digest: sha256:e0036173bdc591b019c3ce4ee1457e423b9a0ff9b3184ee0cbfc5f973ddac6ed
Status: Downloaded newer image for aflplusplus/aflplusplus:latest
docker.io/aflplusplus/aflplusplus:latest
root@debian2ver1210:/home/pavel#
```

Docker создает и запускает контейнер на основе образа AFL++

- t**: Создает псевдотерминал (tty), что позволяет взаимодействовать с контейнером через терминал.
- i**: Позволяет взаимодействовать с контейнером в интерактивном режиме, сохраняя стандартный ввод (stdin) открытым.
- ~/aflt**: Это путь к локальной директории на вашем компьютере, где могут находиться исходные файлы или данные, которые вы хотите использовать для фаззинга.
- /src**: Это путь внутри контейнера, куда будет смонтирована локальная директория. Таким образом, все файлы из **~/aflt** будут доступны в контейнере по пути **/src**.

```
root@debian2ver1210:/home/pavel# sudo docker run -ti -v ~/aflt:/src aflplusplus/aflplusplus
[AFL++ 83f40a912ac7] /AFLplusplus #
```

Далее напишу небольшой код на C для запуска фаззинга

```
GNU nano 7.2                                pavel@debian2ver1210: ~
vulnerable.c
#include <stdio.h>
#include <string.h>

void vulnerable(char *input) {
    char buffer[16];
    strcpy(buffer, input);
}

int main(int argc, char **argv) {
    if(argc > 1) {
        vulnerable(argv[1]);
    }
    return 0;
}
```

И внесу изменения в входной файл

afl-fuzz: Это основной исполняемый файл инструмента AFL++, который отвечает за фаззинг. Он генерирует случайные или мутационные входные данные и отправляет их в целевую программу для тестирования.

```
[AFL++ 6817b82c0fb4] /src/input # nano inputs.txt
[AFL++ 6817b82c0fb4] /src/input # afl-fuzz -i input -o output -- ./main @@
```

```
pavel@debian2ver1210: ~  
  
american fuzzy lop ++4.33a {default} (./main) [explore]  
process timing | overall results  
  run time : 0 days, 0 hrs, 0 min, 11 sec | cycles done : 37  
  last new find : none yet (odd, check syntax!) | corpus count : 4  
last saved crash : none seen yet | saved crashes : 0  
last saved hang : none seen yet | saved hangs : 0  
cycle progress | map coverage  
  now processing : 1.227 (25.0%) | map density : 12.50% / 12.50%  
  runs timed out : 0 (0.00%) | count coverage : 449.00 bits/tuple  
stage progress | findings in depth  
  now trying : havoc | favored items : 1 (25.00%)  
  stage execs : 99/100 (99.00%) | new edges on : 1 (25.00%)  
  total execs : 22.8k | total crashes : 0 (0 saved)  
  exec speed : 2033/sec | total tmouts : 0 (0 saved)  
fuzzing strategy yields | item geometry  
  bit flips : 0/0, 0/0, 0/0 | levels : 1  
  byte flips : 0/0, 0/0, 0/0 | pending : 0  
  arithmetics : 0/0, 0/0, 0/0 | pend fav : 0  
  known ints : 0/0, 0/0, 0/0 | own finds : 0  
  dictionary : 0/0, 0/0, 0/0, 0/0 | imported : 0  
  havoc/splice : 0/22.7k, 0/0 | stability : 100.00%  
  py/custom/rq : unused, unused, unused, unused |  
  trim/eff : n/a, n/a | [cpu:100%]  
strategy: explore | state: started :-)
```

```
pavel@debian2ver1210: ~  
american fuzzy lop ++4.33a {default} (./main) [explore]  
process timing | overall results  
  run time : 0 days, 0 hrs, 41 min, 14 sec | cycles done : 3902  
  last new find : none yet (odd, check syntax!) | corpus count : 4  
last saved crash : none seen yet | saved crashes : 0  
last saved hang : none seen yet | saved hangs : 0  
cycle progress | map coverage  
  now processing : 1.23416 (25.0%) | map density : 14.29% / 14.29%  
  runs timed out : 0 (0.00%) | count coverage : 449.00 bits/tuple  
stage progress | findings in depth  
  now trying : havoc | favored items : 1 (25.00%)  
  stage execs : 76/100 (76.00%) | new edges on : 1 (25.00%)  
  total execs : 2.34M | total crashes : 0 (0 saved)  
  exec speed : 990.0/sec | total tmouts : 47 (0 saved)  
fuzzing strategy yields | item geometry  
  bit flips : 0/0, 0/0, 0/0 | levels : 1  
  byte flips : 0/0, 0/0, 0/0 | pending : 0  
  arithmetics : 0/0, 0/0, 0/0 | pend fav : 0  
  known ints : 0/0, 0/0, 0/0 | own finds : 0  
  dictionary : 0/0, 0/0, 0/0, 0/0 | imported : 0  
  havoc/splice : 0/2.34M, 0/0 | stability : 100.00%  
  py/custom/rq : unused, unused, unused, unused |  
  trim/eff : n/a, n/a | [cpu:300%]  
strategy: exploit | state: finished...
```

В случае краша система об этом скажет:

american fuzzy lop ++4.33a {default} (./main) [explore]	
process timing	overall results
run time : 0 days, 0 hrs, 11 min, 58 sec	cycles done : 10.4k
last new find : none yet (odd, check syntax!)	corpus count : 1
last saved crash : 0 days, 0 hrs, 11 min, 58 sec	saved crashes : 1
last saved hang : none seen yet	saved hangs : 0
cycle progress	map coverage
now processing : 0.31217 (0.0%)	map density : 14.29% / 14.29%
runs timed out : 0 (0.00%)	count coverage : 449.00 bits/tuple
stage progress	findings in depth
now trying : havoc	favorited items : 1 (100.00%)
stage execs : 86/100 (86.00%)	new edges on : 1 (100.00%)
total execs : 3.12M	total crashes : 83.3k (1 saved)
exec speed : 4264/sec	total tmtouts : 4 (0 saved)
fuzzing strategy yields	item geometry
bit flips : 0/0, 0/0, 0/0	levels : 1
byte flips : 0/0, 0/0, 0/0	pending : 0
arithmetics : 0/0, 0/0, 0/0	pend fav : 0
known ints : 0/0, 0/0, 0/0	own finds : 0
dictionary : 0/0, 0/0, 0/0, 0/0	imported : 0
havoc/splice : 1/3.12M, 0/0	stability : 100.00%
py/custom/rq : unused, unused, unused, unused	
trim/eff : n/a, n/a	
[cpu000: 50%]	
strategy: explore	state: finished...

```
pavel@debian2ver1210: ~
pavel@debian2ver1210:~$ su root
Пароль:
root@debian2ver1210:/home/pavel# cd ~/aflt
root@debian2ver1210:~/aflt# ls
input inputs inputs.txt main out output outputs output.txt vulnerable.c
root@debian2ver1210:~/aflt# cd output
root@debian2ver1210:~/aflt/output# ls
default
root@debian2ver1210:~/aflt/output# cd default
root@debian2ver1210:~/aflt/output/default# ls
cmdline fastresume.bin fuzzer_setup hangs queue
crashes fuzz_bitmap fuzzer_stats plot_data target_hash
root@debian2ver1210:~/aflt/output/default#
```

Описание компонентов фаззера

1. **Очередь (queue):** Это место, где хранятся входные данные, используемые фаззером для создания новых тестовых вариантов.
2. **Сбои (crashes):** Каталог, в который сохраняются файлы, вызвавшие сбои программы во время тестирования.

3. **Зависания (hangs):** Папка, где хранятся тестовые файлы, которые привели к зависанию программы, то есть к длительному выполнению без завершения.
4. **Статистика фаззера (fuzzer_stats):** Файлы, содержащие статистические данные о работе фаззера, такие как количество выполненных циклов и покрытие кода.
5. **Данные для графиков (plot_data):** Информация, используемая для построения графиков, например, зависимость покрытия кода от времени.
6. **Хэш целевых файлов (target_hash):** Данные о хэшировании целевых файлов, которые помогают отслеживать уникальность входных данных.
7. **История команд (cmdline):** Запись команд, использованных для запуска фаззера, включая аргументы командной строки.
8. **Битмап покрытия (fuzz_bitmap):** Данные, которые отслеживают покрытие кода, включая битмапы, используемые для определения новых ветвей выполнения.
9. **Настройки фаззера (fuzzer_setup):** Конфигурационные файлы или параметры, определяющие настройки работы фаззера.

Фаззинг является мощным методом тестирования. Его автоматизация и способность находить ошибки делают его незаменимым инструментом в современном процессе разработки программного обеспечения.

Статический анализ кода

Статический анализ кода — это метод анализа программного обеспечения, который позволяет находить ошибки и уязвимости в исходном коде без его выполнения. Этот подход используется для повышения качества и безопасности программ, особенно в больших и сложных системах.

Svace — это один из инструментов статического анализа, разработанный Институтом системного программирования РАН. Он предназначен для обнаружения критических ошибок в исходном коде и поддерживает множество языков программирования, таких как C/C++, Java, C#, Python и другие.

Svaceg — это сервер, предназначенный для хранения и обработки результатов статического анализа кода. Он поддерживает импорт данных из статического анализатора Svace и других инструментов

Svace начинает с мониторинга процесса сборки анализируемой программы. Он перехватывает вызовы компиляторов и запускает собственные модифицированные версии компиляторов для генерации внутреннего представления кода. Это позволяет анализатору получить точные данные о структуре программы и ее зависимости.

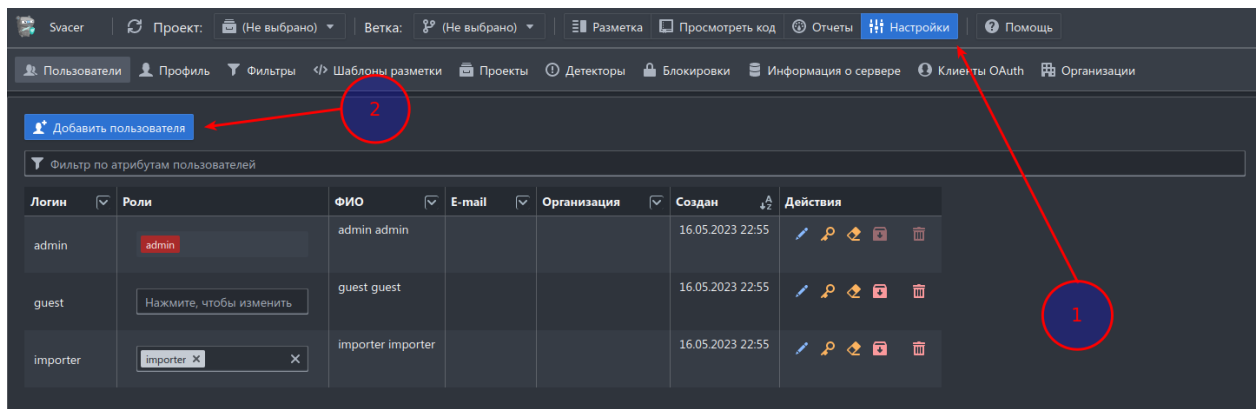
После сборки кода Svaser выполняет статический анализ.

На первом этапе запускаются детекторы, которые ищут ошибки кодирования, специфичные для каждого языка программирования.

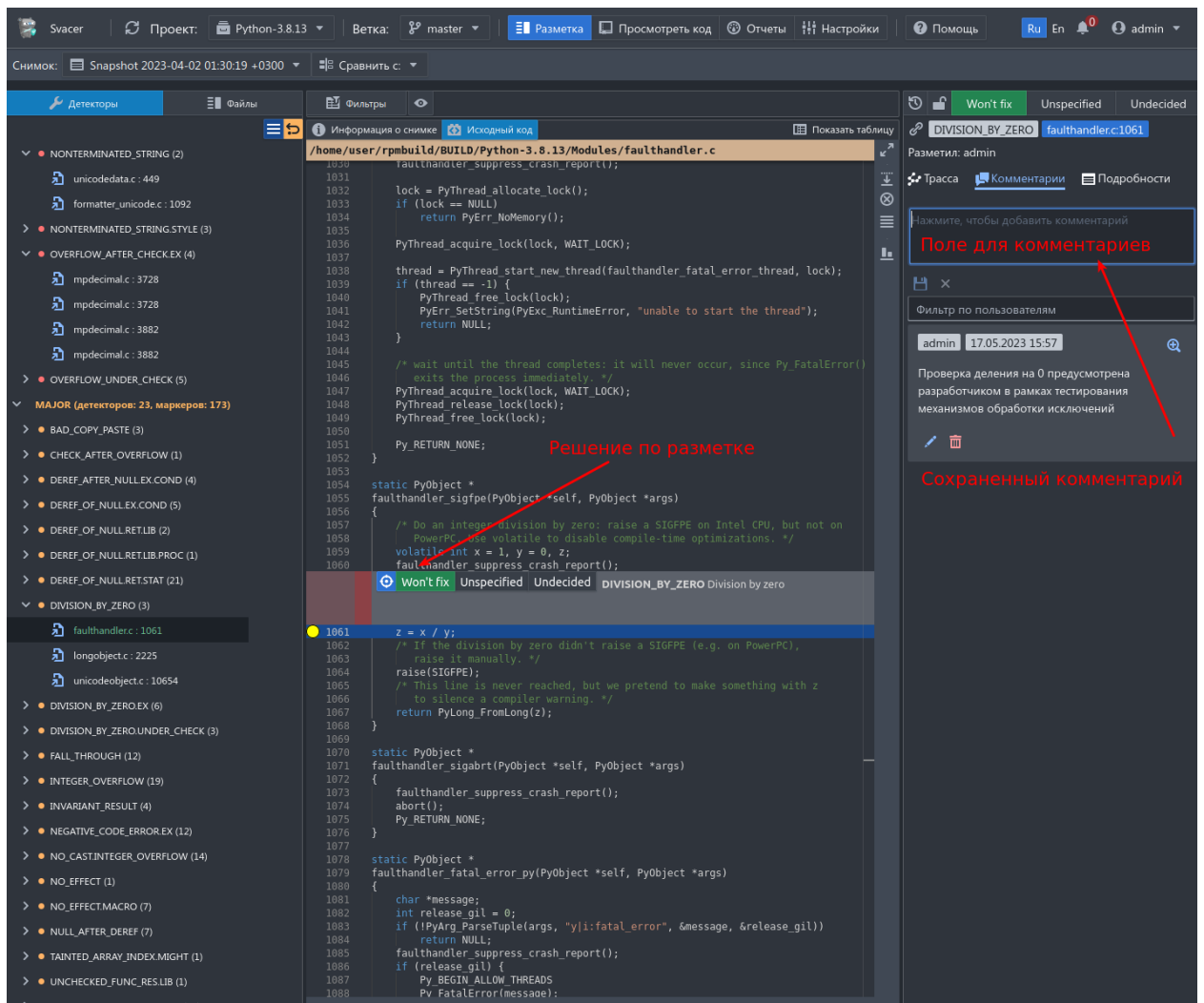
Затем запускается основной движок анализа, который строит граф вызовов программы и выполняет межпроцедурный анализ. Этот анализ осуществляется с использованием резюме функций, что позволяет избежать повторного анализа уже обработанных функций. Резюме содержит информацию о параметрах, возвращаемых значениях и побочных эффектах, что значительно ускоряет процесс анализа.

Результаты анализа сохраняются в базе данных, и пользователи могут просматривать их через веб-интерфейс. В этом интерфейсе можно разметить предупреждения как истинные или ложные, а также отслеживать историю разметки между запусками анализа. Это позволяет улучшать точность анализа и снижать количество ложных срабатываний.

Создание пользователя



В отчете можно выносить решения по уязвимостям и писать к ним комментарии



Таким образом, Svaser представляет собой мощный инструмент для статического анализа кода, который помогает разработчикам находить и устранять ошибки на ранних стадиях разработки, повышая качество и безопасность программного обеспечения.