УДК: 004.056.57 DOI: 10.53816/20753608_2022_3_129

ИССЛЕДОВАНИЕ ТЕХНОЛОГИЙ ПОВЫШЕНИЯ ДОВЕРИЯ К СПЕЦИАЛЬНОМУ ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ С ПРИМЕНЕНИЕМ ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ, РЕАЛИЗУЮЩИХ ФАЗЗИНГ-ТЕСТИРОВАНИЕ

RESEARCH OF TECHNOLOGIES FOR INCREASING CONFIDENCE IN SPECIAL SOFTWARE USING TOOLS THAT IMPLEMENT FUZZING TESTING

По представлению чл.-корр. РАРАН А.В. Грудзинского

А.В. Морозов, Г.Е. Панамарев

Военный инновационный технополис «ЭРА»

A.V. Morozov, G.E. Panamarev

В статье описаны результаты прикладных исследований разработки испытательного стенда для повышения доверия к программному обеспечению с применением инструментальных средств, реализующих фаззинг-тестирование. Обозначенная научно-техническая проблема в настоящее время не имеет окончательного решения. Испытательный стенд позволяет проводить динамический анализ специального программного обеспечения и выявлять программные дефекты и уязвимости непосредственно во время работы программы. Проведенное исследование показало, что применение фаззинга позволяет повысить безопасность программного обеспечения путем выявления уязвимостей в ядре операционной системы Astra Linux.

Ключевые слова: испытательный стенд, фаззинг-тестирование, ядро ОС GNU/Linux, уязвимости в программном обеспечении, повышение доверия к программному обеспечению, динамическое тестирование.

Paper describes research results on development of a testbed for software trustworthiness evaluation based on fuzzing testing. The designated scientific and technical problem currently has no final decision. The proposed testbed allows for dynamic analysis of special purpose software and runtime detection of flaws and vulnerabilities. The conducted research has shown that the use of fuzzing can improve software security by identifying vulnerabilities in the core of the Astra Linux operating system.

Keywords: testbed, fuzzing-testing, GNU/Linux kernel, software vulnerabilities, software trust evaluation, dynamic testing.

Введение

Разработка безопасного системного программного обеспечения (ПО), на основе которого строятся сертифицированные средства защиты информации, достижение доверия к нему согласно требованиям нормативных документов отечественных регуляторов — крупная научно-техническая проблема [1]. Проблема выявления уязвимостей в программных средствах не нова, но она не получила своего исчерпывающего решения и в настоящее время остается чрезвычайно востребованной [2–5]. Собственно наличие уязвимостей в программном обеспечении

составляет основной класс угроз современных компьютерных систем, сетей [6–8]. Долгое время считалось, что динамический анализ ПО является слишком тяжеловесным подходом к обнаружению программных дефектов и полученные результаты не оправдывают затраченных усилий и ресурсов. Однако две важные тенденции развития современной индустрии производства ПО позволяют по-новому взглянуть на эту проблему. С одной стороны, при постоянном увеличении объема и сложности ПО любые автоматические средства обнаружения ошибок и контроля качества могут оказаться полезными и востребованными. С другой — непрерывный рост производительности современных вычислительных систем позволяет эффективно решать все более сложные вычислительные задачи. Особенностью динамического анализа является то, что процесс осуществляется во время работы программы. Фаззинг — методика тестирования, при которой на вход программы подаются невалидные, непредусмотренные или случайные данные.

Основная часть

Проведение фаззинг-тестирования ядра операционной системы (ОС) GNU/Linux включает шесть этапов: сборку тестируемого ядра ОС GNU/Linux; проверку на стабильность собранного ядра ОС GNU/Linux; фаззинг-тестирование ядра ОС GNU/Linux; интерпретацию найденных падений и определение степени угрозы для ОС; воспроизводство найденных падений, для подтверждения того, что не было ложного срабатывания инструментального средства; формирование протокола проведения испытаний.

Для оценки результатов фаззинг-тестирования использовались два вида показателей: количественные и качественные. К количественным показателям, подлежащим оценке в процессе испытаний, относятся процент покрытия кода и количество найденных падений. К качественным показателям относятся: сгенерированные входные данные; уникальность найденных падений.

Выбор инструментальных средств осуществлялся среди наиболее популярных и часто используемых фаззеров. В табл. 1 представлен сравнительный анализ инструментальных средств, реализующих фаззинг-тестирование.

Для обоснованного выбора инструментальных средств проведения фаззинг-тестирования в интересах повышения доверия к ПО выбраны следующие критерии оценки: язык программирования (ЯП); назначение; платформа; наличие документации.

По результатам проведенного анализа [9] было выявлено, что для исследования вопросов повышения доверия к ПО наиболее подходящи-

Таблица 1

Сравнительный анализ инструментальных средств

№ п/п	Наименование	ЯП	Назначение	Платформа	Наличие документации
1	OWASP JBroFuzz	У	У	У	НУ
2	Bunny the Fuzzer	У	ЧУ	У	НУ
3	SPIKE	НУ	ЧУ	У	НУ
4	PFF	НУ	ЧУ	У	НУ
5	ProxyFuzz	НУ	У	У	НУ
6	IOCTL Fuzzer	У	ЧУ	У	НУ
7	zzuf	У	У	У	НУ
8	ИСП Фаззер	У	У	У	У
9	AFL, WinAFL	У	ЧУ	У	НУ
10	LibFuzzer	У	ЧУ	У	НУ
11	Syzkaller	У	У	У	У
12	Peach	У	У	У	ЧУ
13	Avalanche	У	ЧУ	У	НУ
14	KLEE	У	У	У	НУ

ми инструментальными средствами являются «Syzkaller» и «ИСП Фаззер», так как данные инструментальные средства позволяют проводить тестирование и поиск ошибок в ядре операционных систем, а также отдельных компонентов программного обеспечения.

Испытательный стенд (ИС) предназначается для исследования технологий повышения доверия к программному обеспечению с применением инструментальных средств, реализующих фаззинг-тестирование [10]. Состав стенда представлен в табл. 2. Характеристики серверного оборудования представлены в табл. 3.

На сервере была установлена среда виртуализации VMware ESXi. Развернуты пять виртуальных машин (Linux Debian 10, Windows 7, Ubuntu). На ОС Linux Debian 10 установлено программное обеспечение: компилятор gcc-8.3.0, язык программирования go 1.16.4; инструментальное средство «syzkaller» версия с GitHub от 16.06.2021. Характеристики ПЭВМ оператора представлены в табл. 4.

Фаззинг-тестирование программы — это вид работ по исследованию программы, направленный на оценку ее свойств, поиск ошибок и уязвимостей, основанный на передаче программе случайных или специально сформированных входных данных, отличных от данных, предусмотренных алгоритмом работы программы [1].

Таблица 2

Состав стенда

Наименование	Количество
ПЭВМ	4
Монитор	4
Компьютерная периферия (мышь)	4
Компьютерная периферия (клавиатура)	4
Источник бесперебойного питания	4

Таблица 3

Характеристики серверного оборудования

Наименование	Значение
ЦПУ	20 ЦПУ x Intel(R) Xeon(R) Silver 4114 CPU @ 2.20 ГГц
Память RAM	127.66 ГБ
Память HDD	1.78 ТБ (Raid 5: 8x300 ГБ)

Таблица 4

Характеристики ПЭВМ оператора

Наименование	Значение
ЦПУ	Intel i7-6700T
Память RAM	16 ГБ
Память HDD	1 ТБ
Память SSD	128 ГБ

В ходе работы стенда осуществляется фаззинг-тестирование ядра операционной системы Debian. Результатами данного тестирования являются лог-файлы, которые указывают на различные ошибки, которые приводили к падениям либо неправильной работе операционной системы. Найденные ошибки можно воспроизвести по полученным лог-файлам. Некоторые ошибки поддаются воспроизведению, тогда это указывает на наличие потенциальной уязвимости в ядре ОС. Если ошибки не поддаются воспроизведению, то они не представляют интереса, так как они могут считаться ложными срабатываниями. Фаззинг-тестирование ядра операционной системы Debian на испытательном стенде осуществляется с помощью инструментального средства «syzkaller». На сегодняшний день данным ИС поддерживается тестирование следующих ОС: Akaros, FreeBSD, Fuchsia, gVisor, Linux, NetBSD, OpenBSD, Windows [9].

Алгоритм работы «syzkaller»: оператор запускает и перезапускает виртуальные машины (ВМ), на которых происходит фаззинг-тестирование. Для связи с ними используется ssh подключение, утилита фаззера, работающая в каждой ВМ, передает сгенерированные входные параметры в утилиту исполнения, которые формируются в системные вызовы и направляются в ядро ОС. Результаты выполнения и информация о покрытия кода передаются в утилиту фаззера. Все найденные падения во время тестирования и системные вызовы хранятся в рабочей директории. Пользователь просматривает результаты тестирования на веб-странице через «http» подключение. На рис. 1 представлена схема работы «syzkaller».

Для запуска «syzkaller» необходимо дополнительное ПО: средство виртуализации, компилятор языка программирования GO, «openssh», компилятор «GCC» версии 8.3.0 или новее. Фаззинг рекомендуется проводить с включенными детекторами возникновения внештатных ситуаций: санитайзеры и специальные опции, включенные в процессе сборки тестируемого ядра.

Инструментальное средство «syzkaller» позволяет не производить полное тестирование

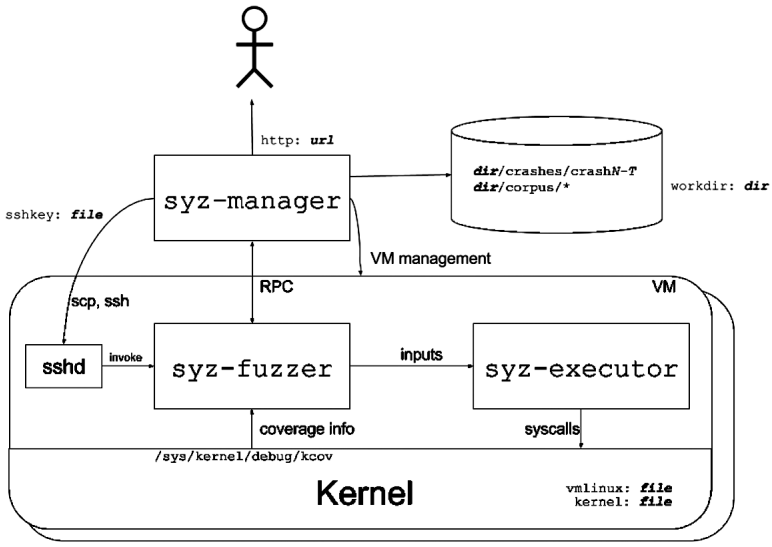

Рис. 1. Схема работы ИС «syzkaller»

ядра, а вместо этого выборочно тестировать отдельные файлы, функции, системные вызовы и каталоги ядра операционной системы. Для проведения фаззинг-тестирования с фильтрацией необходимо внести изменения в конфигурационный файл ИС. Код файла конфигурации с фильтрацией каталогов, то есть тестирование будет проводиться только на указанных файлах:

```
[syzkaller] cat configs/cover NET.cfg.back
{
“name”: “cover net/sctp”, “target”: “linux/amd64”, “http”: “127.0.0.5:56745”, “workdir”: “/home/usr/linuxCores/gopath/src/github.com/google/syzkaller/workdir/cover_NET”, “kernel obj”: “/root/linux-source-4.19”,
“image”: “/home/usr/linuxCores/custom.img”,
“sshkey”: “/home/usr/linuxCores/custom.id rsa”,
“max crash logs”: 10,
“sandbox”: “none”,
“cover”: true,
“cover filter”:
{
“files”: [
“”net/sctp/”, “net/dccp/”
]
b
“ignores”: [“possible deadlock in ext4 evict inode”,
“possible deadlock in sock release”, “possible deadlock in cleanup net”, “INFO: task hung in corrupted”, “INFO: task hung in hubportinit”, “INFO: task hung in usb get descriptor”
L
“reproduce”: false,
“syzkaller”: “/home/usr/linuxCores/gopath/src/github.com/google/syzkaller”,
“procs”: 8,
“type”: “qemu”,
“vm”: {
“count”: 4,
“kernel”: “/root/linux-source-4.19/arch/x86_64/boot/bzImage”,
«cpu»: 2,
«mem»: 2048
}
```

В нем также указано игнорирование уже ранее найденных падений, не имеющих интереса для исследований. По результатам выявленных ошибок и падений составляется протокол. Далее меняется конфигурация ядра, с целью избавления от выявленных ошибок и падений.

Следующий этап — установка и настройка программного обеспечения. Перед установкой программного обеспечения необходимо выполнить сборку ядра с установкой определенных флагов для тестирования ядра ОС GNU/Linux. Рекомендуемые флаги можно посмотреть в документации ИС «syzkaller» по пути «docs/linux/kernel_configs». Для сборки ядра необходимо изменить конфигурацию ядра на хостовой ПЭВМ после распаковки исходного кода, выполнив команду «make menuconfig» в каталоге ядра.

Для сборки ядра необходимо выполнить команду make. После конфигурирования и сборки ядра необходимо установить ПО для проведения фаззинг-тестирования ядра ОС: компилятор языка GO (версии не ниже 1.14.2); инструментальное средство «syzkaller»; компилятор для UNIX-подобных ОС — GCC (версии не ниже 8.3.0) и средство для эмуляции QEMU.

Для того чтобы обновлять репозитории локально, необходимо подключить (примонтировать) 4 образа: Debian-10.9.0-amd64-DVD-1.iso, Debian-10.9.0-amd64-DVD-2.iso, Debian-10.9.0-amd64-DVD-3.iso, Debian-10.9.0-amd64-DVD-4.iso.

Для начала необходимо размесить образы в папке /home/user/iso. Далее нужно прописать данные образы в файле «.bashrc». Это нужно для того, чтобы впоследствии не прописывать полный путь к данным образам. Выполнить команду: sudo nano .bashrc. В конце открывшегося файла прописать следующее:

alias mount1=”sudo mount /home/user/iso/Debian-10.9.0-amd64-DVD-1.iso /media/cdrom0”;
alias mount2=”sudo mount /home/user/iso/Debian-10.9.0-amd64-DVD-2.iso /media/cdrom0”;
alias mount3=”sudo mount /home/user/iso/Debian-10.9.0-amd64-DVD-3.iso /media/cdrom0”;
alias mount4=”sudo mount /home/user/iso/Debian-10.9.0-amd64-DVD-4.iso /media/cdrom0”.

Теперь при выполнении команд mount1/mount2/… будет происходить монтирование соответствующих командам образов. После того как все образы были прописаны и смонтированы, их необходимо размонтировать. Для размонтирования образов необходимо выполнить команду: sudo umount /media/cdrom0.

Для дальнейшей работы необходимо сделать записи об образах в файле source.list. Для автоматической записи в source.list необходимо примонтировать образы по одному (предварительно их все необходимо размонтировать) командой mount1, mount2 и так далее. После подключения каждого образа необходимо выполнять команду sudo apt-cdrom add -m /media/cdrom0. После выполнения данной команды необходимо сначала размонтировать текущий образ, и только после этого монтировать следующий.

Далее зайти в source.list (sudo nano /etc /apt/source.list), изменить строчку, которая начинается со слова deb и сразу после нее вставить: [trusted=yes]. Всего должно получиться 4 такие строчки (по одной на образ). Результат редактирования файла «source.list»:

deb [trusted=yes] cdrom: [Debian GNU/Linux 10.9.0 _ Buster_ - official amd 64 DVD Binary-4 20210327 — 10:39]/ buster contrib main
deb [trusted=yes] cdrom: [Debian GNU/Linux 10.9.0 _ Buster_ - official amd 64 DVD Binary-3 20210327 — 10:39]/ buster contrib main
deb [trusted=yes] cdrom: [Debian GNU/Linux 10.9.0 _ Buster_ - official amd 64 DVD Binary-2 20210327 — 10:39]/ buster contrib main
deb [trusted=yes] cdrom: [Debian GNU/Linux 10.9.0 _ Buster_ - official amd 64 DVD Binary-1 20210327 — 10:39]/ buster contrib main

Теперь можно пользоваться данными образами для обновления репозиториев.

Установка языка программирования GO. Язык программирования GO необходим для работы фаззера «syzkaller». Архив с GO необходимо переместить в предварительно созданную папку по пути: /home/user/soft_for_fuzzing. После этого выполнить следующие действия: tar -xf /home/user/soft_for_fuzzing/go 1.16.2.linux-amd64.tar.gz — распакует архив с ЯП GO; mv go goroot — переместит распакованный архив в каталог goroot; mkdir gopath — создаст каталог; export GOPATH =’pwd’/gopath — экспортирует пути для GO; export GOROOT =’pwd’/goroot; export PATH = $GOPATH/bin:$PATH; export PATH = $GOROOT/bin:$PATH.

Можно строго указать экспортированные пути в файле «.bashrc», иначе придется указывать их каждый раз при запуске терминала. Для этого нужно зайти в файл «.bashrc». В открывшемся файле прописать: export GOPATH =/home/user/gopath; export GOROOT =/home/user/goroot; export PATH =$GOPATH/bin:$PATH; export PATH =$GOROOT/bin:$PATH. На этом установку языка программирования GO можно считать оконченной.

Установка «syzkaller», «gcc», «debootstrap». Архив с «syzkaller» распаковать в папку по пути: /home/user/gopath/src/github.com/google. Предварительно создать каталоги src, github.com, google командой: mkdir -p src/github.com/google. Далее переименовать распакованный в папку google архив: mv syzkaller-master syzkaller. После этого, зайти в переименованный каталог syzkaller, после чего выполнить инициацию git — репозитория: git init. Выполнить git add — данная команда добавит все файлы из каталога в репозиторий git. Выполнить закрепление изменений в данном репозитории: git commit-m «Тут можно написать комментарий».

Дальше выполнить команду make generate && make, она должна сгенерировать все GO файлы и исполняемые утилиты. Команду generate следует использовать в том случае, если был отредактирован исходный код ИС «syzkaller». В случае возникновения ошибок с зависимостями, необходимо их устранить или же устанавливать по мере необходимости: sudo apt install gcc, clang-format, g++.

Debootstrap — утилита для создания формирования образа базовой системы в подкаталог файловой системы запущенной ОС. Далее этот каталог будет именоваться chroot. Сформированная система в каталоге используется для создания образа, который необходим для запуска ВМ в «QEMU» и тестирования с помощью ИС «syzkaller».

В процессе установки пакетов могут потребоваться дополнительные пакеты, которые находятся на других репозиториях. После подключения необходимого репозитория, нужно набрать команду fg, чтобы вернуться в приостановленный процесс установки и нажать enter для продолжения и завершения установки.

Для проверки работоспособности собранного образа с помощью утилиты «debootstrap» необходимо воспользоваться средством «QEMU» для эмуляции аппаратного обеспечения. Чтобы среда виртуализации «QEMU» заработала в виртуальной машине, необходимо включить параметры: «Hardware virtulization» и «I/O MMU» в настройках виртуальной машины, в данном случае в VMware ESXi. Для создания образа, используемого для эмуляции в «QEMU» используется скрипт:

```
/bin/bash
set -eux
#Create a minimal Debian distribution in a directory.
DIR=chroot
SEEK-2647
HAME=testl
#Set some defaults and enable promtless ssh to the machine for root,
sudo sed -i '/^root/ { s/:x:/::/ }' chroot/etc/passwd
echo 'T0:23:respawn:/sbin/getty -L tty50 115200 vtl00' | sudo tee -a chroot/etc/inittab
printf '\nauto eth0\niface eth0 inet dhcp\n' | sudo tee -a chroot/etc/network/interfaces
echo '/dev/root / ext4 defaults 0 | sudo tee -a chroot/etc/fstab
echo 'debugfs /sys/kernel/debug debugfs defaults 0 0 | sudo tee -a chroot/etc/fstab
echo 'securityfs /sys/kernel/security securityfs defaults 0 0' | sudo tee -a chroot/etc/fstab
echo 'configfs /sys/kernel/config/ configfs defaults 0 0' | sudo tee -a chroot/etc/fstab
echo 'binfmt.misc /proc/sys/fs/binfmt_misc binfmtjmisc defaults 0 0' | sudo tee -a chroot/etc/fstab echo "kernel.printk
= 7 4 1 3" | sudo tee -a chroot/etc/sysctl.conf
echo 'debug.exception-trace = 0' | sudo tee -a chroot/etc/sysctl.conf
echo "net.core.bpf_jit_enable = 1' | sudo tee -a chroot/etc/sysctl.conf
echo "net.core.bpf_jit_kallsyms = 1' | sudo tee -a chroot/etc/sysctl.conf
echo "net.core.bpf_jit_harden = 0' | sudo tee -a chroot/etc/sysctl.conf
echo "kernel.softlockup_all_cpu_backtrace = 1" | sudo tee -a chroot/etc/sysctl.conf
echo "kernel.kptr.restrict - 0" | sudo tee -a chroot/etc/sysctl.conf
```

```
echo “kernel.watchdog_thresh ■ 60” | sudo tee -a chroot/etc/sysctl.conf
echo “net.ipv4.ping_group_range - 0 65535” | sudo tee -a chroot/etc/sysctl.conf
echo -en “127.9.9.l\tlocalhost\n“ | sudo tee chroot/etc/hosts
echo “nameserver 0.0.0.0” | sudo tee -a chroot/etc/resolve.conf
echo “syzkaller” | sudo tee chroot/etc/hostname
ssh-keygen -f SHAME.id.rsa -t rsa -N ”
sudo mkdir -p chroot/root/.ssh/
cat $NAME.id_rsa.pub | sudo tee chroot/root/.s$h/authorized_keys
#Build a disk image
dd if«/dev/zero of-$NAME.img bs-lM seek-$SEEK count-1
sudo mkfs.ext4 -F SNAME.img
sudo mkdir -p /mnt/chroot
sudo mount -o loop SNAME.img /mnt/chroot
sudo cp -a chroot/. /mnt/chroot/.
sudo umount /mnt/chroot
```

Для проверки работоспособности собранного тестируемого образа рекомендуется использовать скрипт:

```
/bin/sh
<ERNEL=/home/user/soft_for_fuzzing/root/linux-source-4.19 [MAGE=/home/user/soft_for_fuzzing
NAME-testl
emu-system-x86_64 \
-kernel $KERHEL/bzImage \
-append “console=tty50 root=/dev/sda debug earlyprint-k=serial slub_debug=QUZ”
-hda $IMAGE/$NAME.img \
-net user,hostfwd=tcp:127.0.0.1:10021-:22 -net nic \
-enable-kvm \
-nographic \
-m 2G \
-smp 2 \
-pidfile vm.pid \
2>&1 | tee vm.log
```

На рис. 2 представлен результат запуска собранного образа системы по представленному скрипту.

Заключение

Рандомизированное тестирование приложений является одним из перспективных методов поиска ошибок. Цель фаззинг-тестирования — определить стабильность приложений при обработке псевдослучайно сгенерированных входных данных. В ходе тестирования приложение запускается на множестве произвольных входных данных, которые могут быть недействительными/неожиданными. Применение фаззинга для генерации разных сценариев использования программного интерфейса приложения и соответствующих входных данных позволяет эффективным образом выявить ошибки в реализации функций программного интерфейса. С помощью стенда решаются следующие задачи: оценка эффективности отечественных средств защиты информации, операционных систем, средств виртуализации, офисных приложений на базе операционной системы Astra Linux; повышение доверия к специальному программному обеспечению, входящему в состав или взаимодействующему с перспективными средствами защиты информации, а также повышение безопасности программного обеспечения путем выявления уязвимостей в ядре операционной системы.

Литература

1. Девянин П.Н., Тележников В.Ю., Хорошилов А.В. Формирование методологии разработки безопасного системного программного обеспечения на примере операционных систем // Труды ИСП РАН. 2021. № 5. С. 25–40.

2. Вареница В.В., Марков А.С., Савченко В.В., Цирлов В.Л. Практические аспекты выявления уязвимостей при проведении сертификационных испытаний программных средств защиты информации // Вопросы кибербезопасности. 2021. № 5 (45). С. 36–44.

3. Марков А.С., Цирлов В.Л. Структурное содержание требований информационной безопасности // Мониторинг правоприменения. 2017. № 1 (22). С. 53–61.

4. Климов С.М., Купин С.В., Антонов С.Г. Оценка защищенности систем передачи дан-

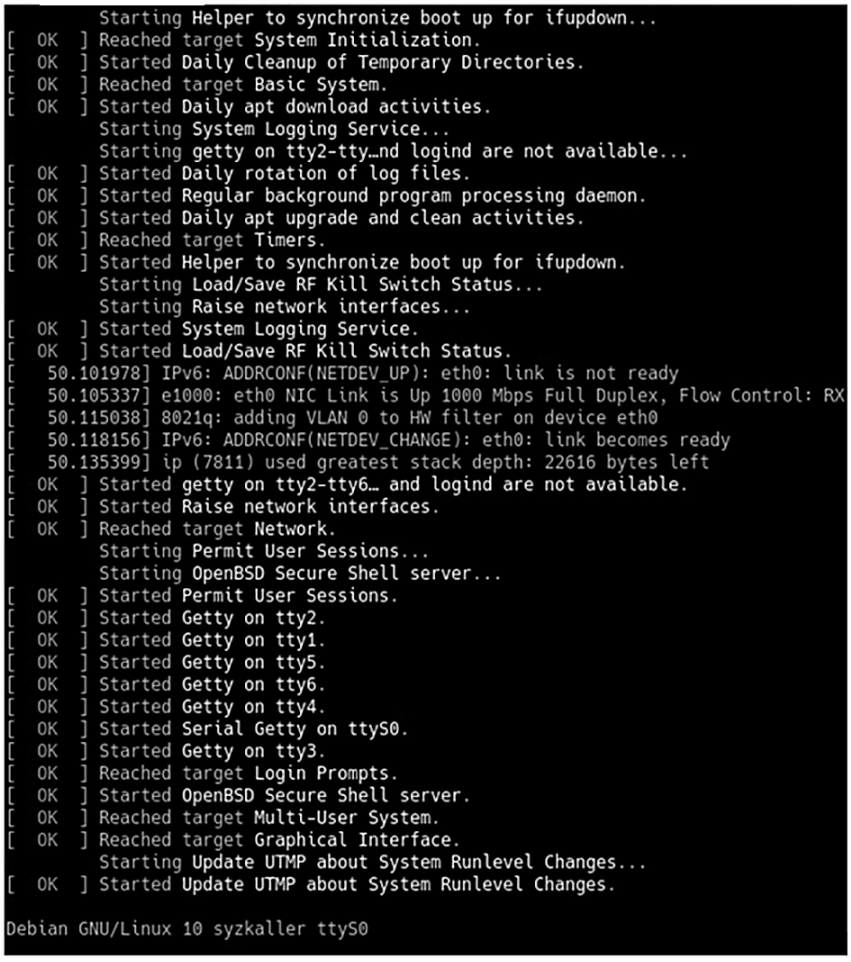

Рис. 2. Проверка работоспособности собранного образа в «QEMU»

ных Вооруженных Сил Российской Федерации // Военная мысль. 2020. № 8. С. 92–96.

5. Томилов И.О., Карманов И.Н., Звягинцева П.А. и др. Разработка методики применения фаззинга для анализа уязвимостей программного обеспечения // Системы управления, связи и безопасности. 2018. № 4. С. 48–63.

6. Барабанов А.В., Евсеев А.Н. Вопросы повышения эффективности анализа уязвимостей при проведении сертификационных испытаний программного обеспечения по требованиям безопасности информации // НиКа. 2015. № 1. С. 330–333.

7. Байдин Г.С., Хизова М.В. Поиск ошибок в программах для обработки графических изображений с использованием метода фаззинга // Вестник МГТУ им. Н.Э. Баумана. Серия «Приборостроение». 2021. № 3 (136). С. 4–23.

8. Шудрак М.О., Золотарев В.В., Лубкин И.А. Методика динамического анализа уязвимостей в бинарном коде // Сибирский аэрокосмический журнал. 2013. № 4 (50). С. 84–87.

9. Поддубный М.И., Кипайкин М.Н., Томилов Д.А. и др. Сравнительный анализ инструментальных средств, реализующих фаззинг-тестирование в интересах повышения доверия к разрабатываемому программному обеспечению // Материалы III Всероссийской НТК «Состояние и перспективы развития современной науки по направлению «АСУ, информационно-телекоммуникационные системы». 2021. Том 2. С. 168–174.

10. Золотарева Е.Ю., Созин М.В. Разработка испытательного стенда для динамического тестирования программного обеспечения // Решетневские чтения. 2016. № 20. С. 256–258.