Как защититься от «бестелесных» веб-шеллов

Авторы: Даниил Садырин, Андрей Сикорский, CyberOK

Введение

В сегодняшней статье эксперты Сайбер ОК проведут вас за руку по лабиринту хакерских уловок и на пальцах объяснят, что такое "бестелесные" веб-шеллы и как с ними бороться. Сегодня мы наблюдаем непрерывную эволюцию кибератак, а в эпицентр этой цифровой метаморфозы встают именно "бестелесные" веб-шеллы – ключевые инструменты злоумышленников для удаленного управления веб-серверами. От обычных они отличаются тем, что не оставляют следов в виде файлов на сервере, что делает их поиск и анализ значительно сложнее.

Мы в CyberOK сталкиваемся с такими кибер-призраками ежедневно и знаем, как переиграть негодяев и защитить свои ресурсы. Вам тоже расскажем в этой статье – мы рассмотрим устройство бестелесных веб-шеллов на примере языка PHP, методы их обнаружения с помощью opensource-инструментов (SOLDR, SELinux), а также посоветуем, как обеспечить безопасность веб-приложений с учетом этих новых угроз. Данное исследование было представлено на конференции PHDays 2023 в секции Fast-Track.

Какие бывают веб-шеллы

Веб-шелл — это командная оболочка для удаленного управления веб-сервером. Рассмотрим, как атакующий может удаленно выполнять команды на сервере, используя веб-приложение:

1. Через заранее созданный файл на диске с кодом на языке программирования и доступным из веба. Недостаток в том, что создается файл на диске, поэтому его легко обнаружить.
2. Выполнять код через баги в веб-приложении (command injection, object injection и др.). Недостаток в том, что запросы могут быть задетектированы Web application firewall (WAF) или баг запатчен разработчиками.
3. Внести изменения в исходный код веб-приложения. Это может быть обнаружено системой контроля версий.
4. Прятать код в базе данных (если там, к примеру, хранятся шаблоны или конфиги). Для этого требуется специальный код в веб-приложении и модификацию базы можно обнаружить.
5. Прятать код в модулях веб-сервера или глобальных конфигах. Для этого требуются высокие права в системе.

Сокрытие в памяти процесса веб-сервера

Но можно прятаться в памяти процесса веб-сервера. Преимущества этого подхода перед ранее перечисленными:

- не нужно использовать файл на диске или базу данных;
- можно маскировать запросы для выполнения кода под запросы к легитимным скриптам.

Недостатки подхода:

- бэкдор в памяти работает до аварийного завершения процесса или перезапуска веб-сервера.

Такие, так называемые «бестелесные» («fileless») веб-шеллы, реализованы для разных языков программирования. В Java они нацелены на внедрение через классы фреймворков Java EE, Spring, Tomcat, в C# внедряются через фреймворк ASP.NET.

Мы же рассмотрим "бестелесный" веб-шелл для языка программирования PHP.

Внедрение в процесс PHP

Из исследований на тему внедрения кода в процесс PHP стоит выделить:

- Выступление на ZeroNights 2018.

Там был предложен запуск ELF-файлов из памяти посредством системного вызова memfd_create. Для вызова memfd_create использовалась запись в файл /proc/self/mem через интерпретатор PHP. Но запись в /proc/self/mem может быть отключена настройками Linux и не работает для mod_php по дефолту.

- Статью исследователей из LogicalTrust.

В ней через баг в PHP внедряется код для перехвата запросов к веб-серверу Apache. Исследование довольно старое и в нем не предлагается удаленного выполнения команд.

- Цикл статей о «Бэкдоринге XAMP-стека».

В статьях предлагается прятать бэкдор в разделяемых библиотеках для PHP, Apache, базы данных. Требуются права на редактирование конфигурационного файла php.ini и перезапуск Apache.

- Множество proof-of-concept-ов по обходу disabled_functions в PHP.

Используются бинарные баги в интерпретаторе PHP для патчинга памяти процесса, чтобы выполнить функцию zif_system в обход ограничений disabled_functions. Этот способ наиболее универсальный, но требует поиска новых багов в PHP.

- Обход disabled_functions через LD_PRELOAD.

В PHP-скрипте задается переменная окружения LD_PRELOAD, содержащая путь до so-файла с кодом. Далее вызывается функция, запускающая какой-либо внешний исполняемый

файл (например, функция mail). Создается новый процесс, который наследует переменные окружения определенные в PHP-скрипте. Заданный в LD_PRELOAD so-файл будет загружен в память нового процесса. Недостаток в том, что создается отдельный процесс и его легко обнаружить.

Ни один из перечисленных подходов не дает универсального способа внедрения «бестелесного» веб-шелла, имея минимальные права на сервере.

Аудит багов в PHP

В процессе пентеста веб-приложений, написанных на PHP, нужно иметь в виду, что баги могут встречаться как в коде веб-приложений, так и в самом языке PHP. Для поиска актуальной информации о багах можно воспользоваться баг-трекером bugs.php.net. За всю историю языка баг-трекер собрал в себе более 80000 записей. Но чтобы результативно разбираться с багами PHP, необходимы некоторые знания о его внутреннем устройстве.

Немного PHP internals

PHP это язык с динамической типизацией. Переменная в PHP представляется структурой zval. Главные поля этой структуры — это type (хранит тип переменной) и zend_value (хранит структуру, соответствующую типу переменной).

PHP имеет собственный менеджер кучи. Освобождённые адреса памяти хранятся в односвязных списках, называемых бинами. В один бин попадают освобождённые адреса памяти одинакового размера. Бины работают по принципу LIFO (last input first output). В PHP7 структура zval не выделяется на куче, в отличие от предыдущих версий языка. Эта структура встраивается в другие более сложные структуры, которые выделяются на куче (например HashTable). Подробнее про PHP zval и менеджер памяти можно прочитать в главах книги PHP Internals Book.

При отладке PHP-багов в gdb можно воспользоваться дополнительными командами из файла .gdbinit, который распространяется вместе с исходниками PHP. Команды упрощают вывод внутренних структур языка в gdb.

История одного бага

Рассмотрим баг с номером #80663. Баг находится в классе SplFixedArray — это класс массива, имеющего фиксированную длину. При изменении размера у объекта класса SplFixedArray лишние элементы удаляются. Баг заключается в том, что при изменении размера массива, посредством вызова метода setSize, в процессе удаления лишних элементов размер массива не пересчитывается на новый. Получается, что в процессе удаления некоторые элементы уже удалены, память для них освобождена, но в коде

деструктора к удаленным элементам можно обратиться, так как размер массива ещё не изменился. Это приводит к багу use-after-free.

Случай, когда новый размер массива равен 0, был пофикшен в баге #80663, но похожий случай, когда новый размер меньше уже имеющегося размера, но при этом больше 0, нет. Данному багу был присвоен номер #81992.

Рассмотрим такой код:

```
<?php
class InvalidDestructor {
    public function __destruct() {
        global $obj;
        $a = str_repeat('A', 100);
        var_dump($obj[2]);
    }
}

$obj = new SplFixedArray(5);
$obj[2] = str_repeat('B', 100);
$obj[3] = new InvalidDestructor();
$obj->setSize(2);
```

```
Expected result:
----------------
string(10) "BBBBBBBBBB"

Actual result:
--------------
string(10) "AAAAAAAAAA"
```

Будет выведена строка из букв BBB вместо ожидаемой строки из букв AAA, потому что память под исходную строку будет переиспользована.

От use-after-free к записи в память процесса

Но как получить из этого бага что-то более полезное? Как насчет чтения/записи в память по произвольному адресу? Снова обратимся к классу SplFixedArray.

Он определен в файле ext/spl/spl_fixedarray.c и представлен структурой _spl_fixedarray.

```
typedef struct _spl_fixedarray { /* {{{ */
    zend_long size;
    zval *elements;
} spl_fixedarray;
```

Поле size в структуре хранит размер массива, поле elements — указатель на память, выделенную на куче под массив из zval нужного размера.

Тут выделяется память для элементов:

```
85          array->elements = ecalloc(size, sizeof(zval));
86          array->size = size;
```

Тут освобождается:

```
199         if (intern->array.size > 0 && intern->array.elements) {
200             efree(intern->array.elements);
201         }
```

Создадим ещё один объект класса SplFixedArray, который освободим вызовом setSize.

После этого используем освобождённую память поля elements для хранения строки (в PHP это структура zend_string).

```
struct _zend_string {
    zend_refcounted_h gc;
    zend_ulong        h;
    size_t            len;
    char              val[1];
};
```

Перекрываем строкой все zval-ы, хранящиеся в освобождённом массиве elements. Для создания zval строки задаем в фейковой структуре zval поле type = 6 и zend_value адресом памяти, откуда хотим прочитать/записать. Обращаемся из PHP-скрипта к созданной нами в памяти строке и пишем в память по заданному адресу, написав в скрипте $this->uaf_obj[0][0] = 'A'.

Разработка proof-of-concept велась для PHP 7.4. В версиях 8.x возможны изменения в PHP internals, портирование под эти версии оставим для заинтересованных.

PHP Life-cycle

Наша цель — это создание «бестелесного» веб-шелла. Для этого надо внедрить код так, чтобы он выполнялся при каждом запросе к веб-серверу от клиента. Чтобы разобраться как это сделать, рассмотрим жизненный цикл PHP, когда он настроен на работу в связке с веб-сервером:

- PHP-процесс при старте загружает в память свои модули;
- при каждом запросе к PHP-скриптам, у загруженных модулей вызывается функция инициализации (RINIT).

Перезаписав адрес функции инициализации (RINIT) в структуре загруженного модуля в памяти на адрес шелл-кода, получим, что шелл-код будет выполняться при каждом запросе к веб-серверу.

Apache может работать в одной из трех моделей обработки запросов:

- mpm_event / mpm_worker — требуют потоко-безопасной версии PHP;
- mpm_prefork — не требует потоко-безопасности.

Nginx почти всегда работает вместе с PHP-FPM, который не требует потоко-безопасности.

В потоко-безопасном PHP каждый поток имеет собственное хранилище данных. Запрос от клиента обрабатывается потоком. В PHP без потоков запрос обрабатывается одним процессом.

PHP, настроенный как модуль Apache (mod_php), требует запуска Apache под root-привилегиями. Из-за настройки ptrace_scope (по умолчанию 1) в этом случае запись в файл /proc/self/mem невозможна, так как владелец дочернего процесса не совпадает с

родительским. В PHP-FPM запись в /proc/self/mem разрешена — это может упросить внедрение «бестелесного» веб-шелла, так как не требуется бинарных багов для записи в память.

Разработка шелл-кода

Необходимо разработать шелл-код для выполнения при каждом запросе от клиента. Сгенерим шаблон проекта дополнения PHP консольной командой:

$ php php-src/ext/ext_skel.php --ext example

В созданном файле example.c пропишем наш код, использующий Zend API, в функцию PHP_RINIT_FUNCTION(example). Код на сервере будем выполнять вызовом zend_eval_string. Необходимо учесть, что адреса функций на сервере заранее неизвестны из-за ASLR, поэтому оставляем для них в шелл-коде место в виде "заглушек", которые потом заполним. Собираем расширение, извлекаем байты функции zm_activate_example из so-файла — это и будет шелл-код. Для этого написан небольшой скрипт на Python. Чтобы быть потоко-безопасным, шелл-код должен вызывать функцию менеджера ресурсов PHP, tsrm_get_ls_cache(), которая вернет область памяти, привязанную к текущему потоку.

Стадии работы эксплойта

Загружаем на сервер PHP-скрипт, например через уязвимость File upload. Скрипт делает следующее:

1. Ищет в памяти шелл-код, который передан в PHP-строке. Для этого используется метод, предложенный в php7-backtrace-bypass эксплойте. Располагаем строки друг за другом в одном бине, перекрываем освобождённую строку объектом класса Helper.

   class Helper { public $a, $b, $c, $d; }

   Читаем адрес next_free_slot и считаем адрес строки-шеллкода, так как строка расположена в том же бине.

2. Читает файл /proc/self/maps, чтобы получить адрес библиотеки PHP.
3. Парсит заголовок ELF-файла библиотеки PHP и её секции.
4. Ищет функции и гаджеты для ROP-цепочки и шелл-кода. Заполняет «заглушки» в шелл-коде адресами функций. Находит адрес загруженного модуля (структура zend_module_entry).
5. Для выполнения ROP, необходимо найти и перезаписать адрес возврата. Идея поиска адреса возврата взята из задания CTF. Посмотрим на backtrace при выполнении любого PHP-скрипта.

```
#3  0x00007f0d16a67bb9 in zend_execute (...) at
#4  0x00007f0d169d7bf1 in zend_execute_scripts
#5  0x00007f0d1696f970 in php_execute_script (.
#6  0x00007f0d16a69d30 in php_handler (...) at
```

6. Функция php_execute_script всегда вызывает zend_execute_scripts, поэтому в стеке всегда будет адрес возврата обратно в php_execute_script. Находим адрес символа php_execute_script, читаем код функции, ищем, где в ней вызывается zend_execute_scripts@plt.

   Важно! Новые версии компиляторов, например GCC 9, входящий в Ubuntu 20.04, помещают код для вызова импортируемых через @plt функций в секцию ".plt.sec". Парсим секцию ".plt.sec", находим в секции запись соответствующую функции zend_execute_scripts. Подробнее про секцию ".plt.sec" можно почитать тут.

7. Ищем посчитанный адрес возврата в стеке. Если имеем дело с потоко-безопасным PHP, то любой поток может обработать запрос от клиента, причем заранее неизвестно какой именно. Поэтому сканируем стеки всех потоков. Размер стека потока в pthreads по умолчанию 8 Мб. Ищем регионы такого размера в /proc/self/maps, сканируем найденные регионы памяти в обратном направлении в поисках адреса возврата. Найдя в стеке адрес возврата, записываем туда ROP-цепочку.

   Внутри ROP-chain:

   1. Вызовом mprotect устанавливаем странице с шелл-кодом права на запись. Шелл-код поместим в незанятое пространство после секции .fini.
   2. Вызовом memcpy копируем туда шелл-код из кучи PHP.
   3. Устанавливаем странице с шелл-кодом права на исполнение.
   4. Перезаписываем указатель RINIT в загруженном модуле на адрес шелл-кода.
   5. Вызываем zend_timeout(0), чтобы корректно завершить PHP-скрипт без Segfault.

   Подробнее про ROP-chains можно почитать тут.

Запускаем PHP-скрипт несколько раз, чтобы забэкдорить все процессы веб-сервера, затем удаляем скрипт. Представленный метод внедрения «бестелестного» веб-шелла подходит для любой конфигурации PHP.

Скачать готовый proof-of-concept можно тут. Папка «docker» содержит Dockerfile и скрипт для развертывания окружения (Apache / PHP 7.4.33 в режиме mod_php) с веб-шеллом. В папке «dev» находятся исходники PHP-расширения.

График исправления бага

2023-05-10	отчёт о баге отправлен на bugs.php.net
2023-05-25	баг принят разработчиками PHP, присвоен #81992
2023-08-14	патч для бага внесен в репозиторий PHP
2023-08-31	вышел PHP 8.2.10 с исправленным багом

- SPL:
  - Fixed bug #81992 (SplFixedArray::setSize() causes use-after-free).

Обнаружение «бестелесных» веб-шеллов

Рассмотрим обнаружение такого рода веб-шелла, используя инструментарий EDR/SELinux.

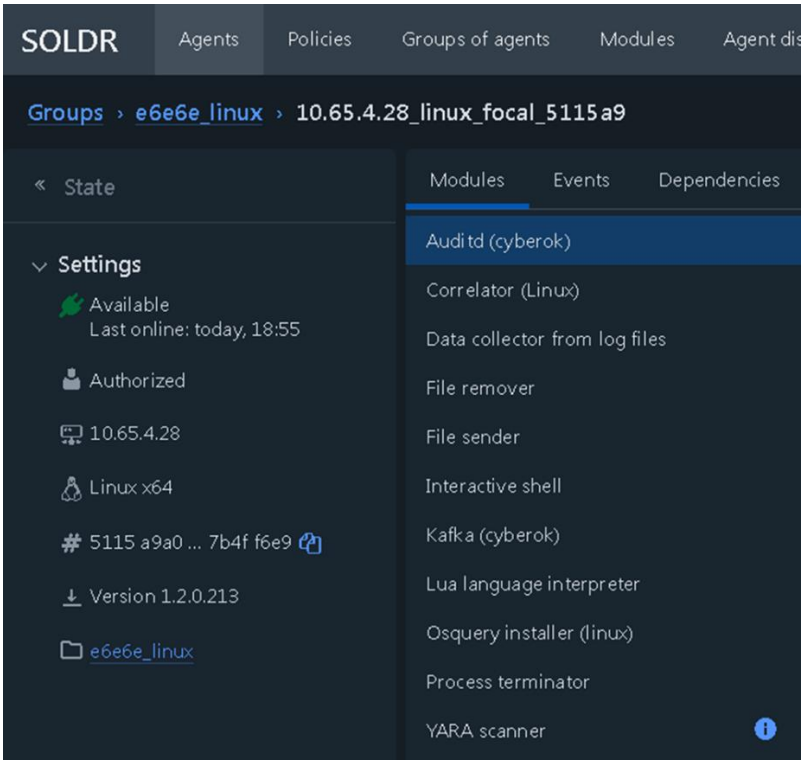
Мы рассматриваем эксплуатацию веб-сервера с PHP — чаще всего он устанавливается на ОС семейства Linux. На Linux узлах обычно не настроены средства мониторинга.

Сосредоточимся на трёх активностях атакующего:

- попытка инъекции кода в процесс веб-сервера;
- уже внедрённый шелл-код в памяти процесса;
- пост-эксплуатация.

Будем использовать SOLDR (System of Orchestration, Lifecycle control, Detection and Response). Это система класса EDR, разрабатываемая компаниями Positive Technologies и CyberOK. С ноября 2022 года этот проект стал открытым, исходный код выложен на GitHub. Агент SOLDR кроссплатформенный, архитектура модульная, у каждого функционального модуля есть своё назначение. Будут полезны следующие модули SOLDR:

- модуль аудита;
- OSquery;
- коррелятор, который получает потоки событий от модулей, нормализует и строит цепочки событий;
- YARA-сканер для поиска вредоносного кода в памяти;
- интерактивный шелл для подключения к узлам с агентом SOLDR.



Детект инъекции кода в процесс

Создадим несколько правил для модуля аудита, чтобы отслеживать подозрительную активность процессов:

- ptrace – используется для внедрения стороннего кода в процесс или режима отладки.

-a always,exit -F arch=b32 -S ptrace -F a0=0x4 -k code_injection

-a always,exit -F arch=b64 -S ptrace -F a0=0x4 -k code_injection

- memfd_create – используется для создания анонимного файла в памяти, не задействуя файловую систему.

-a always,exit -F arch=b64 -S memfd_create -F -k memfd

-a always,exit -F arch=b32 -S memfd_create -F -k memfd

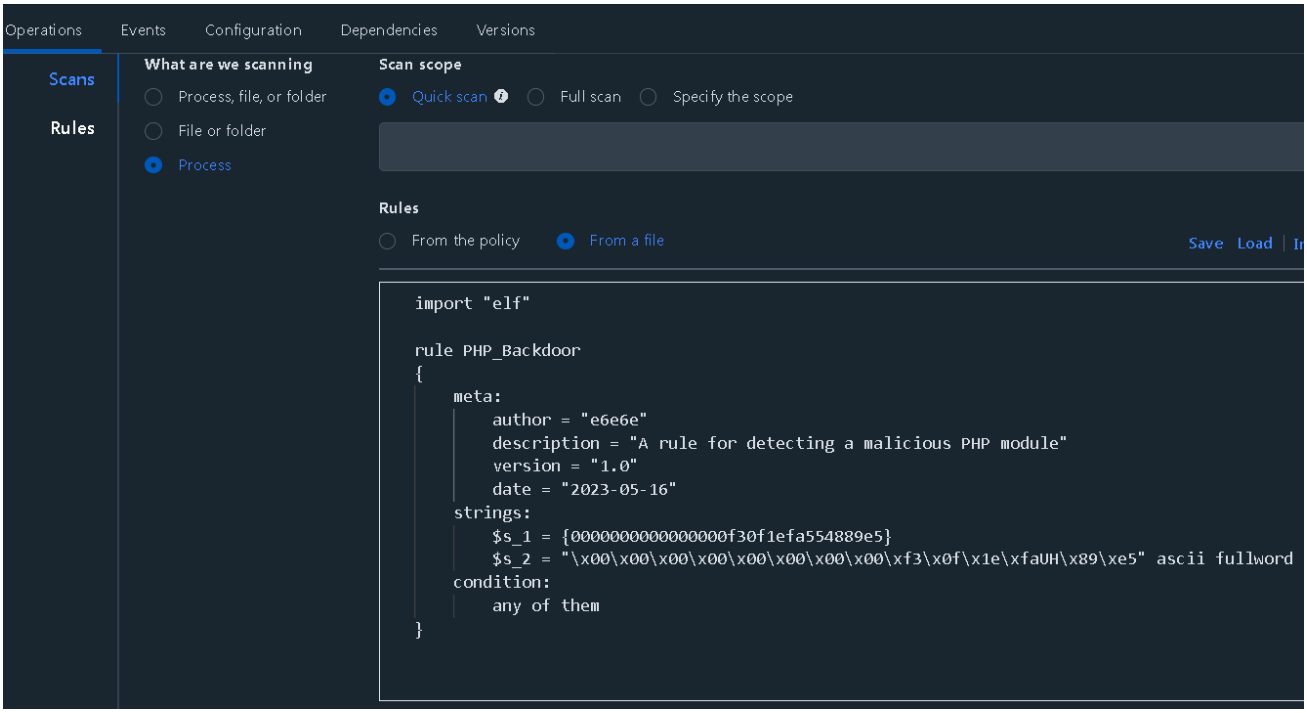
- mprotect – задает права доступа к страницам памяти. Отслеживаем вызовы только от процессов под учетной записью веб-приложения (euid=33). Используется как в легитимной работе веб-приложения, так и для смены прав доступа на память атакующим.

-a always,exit -F arch=b64 -F euid=33 -S mprotect -k www_mprotect

-a always,exit -F arch=b32 -F euid=33 -S mprotect -k www_mprotect

Детект уже внедренного шелл-кода

Если код уже был загружен в процесс веб-сервера, можно воспользоваться модулем YARA-сканера и просканировать процессы веб-сервера в оперативной памяти на наличие сигнатуры шелл-кода. Надо написать своё YARA-правило и запустить скан.



Скан памяти можно запускать как вручную, так и по триггеру, котором может выступать событие от любого SOLDR-модуля.

Можно с помощью запросов OSquery искать регионы памяти с правами rwx и не принадлежащие файлу. В них, вероятно, и будет запрятан шелл-код.

SELECT processes.name, process_memory_map.*, pid as mpid from process_memory_map join processes USING (pid) WHERE process_memory_map.permissions = 'rwxp' AND process_memory_map.path = '';

Детект пост-эксплуатации

Для детекта пост-эксплуатации также поможет модуль аудита. Будем мониторить создание новых процессов, файловую активность, сетевую активность, исходящую от процессов под учетной записью веб-сервера.

Правила аудита:

- Создание новых процессов

  -a always,exit -F arch=b32 -S execve -F euid=33 -k execve_www

  -a always,exit -F arch=b64 -S execve -F euid=33 -k execve_www

- Файловая активность

  -a never,exit -F arch=b64 -F path=<legitimate_path> -F euid=33

  -a always,exit -F arch=b64 -F path=/ -F euid=33 -k fileaccess_www

В правиле в первой строке должны быть исключены те директории и файлы, для которых активность является легитимной. Потом регистрируем все остальные события.

- Сетевая активность

  -a always,exit -F arch=b64 -S socket -F exe=/opt/apache2/bin/httpd -k socket_www

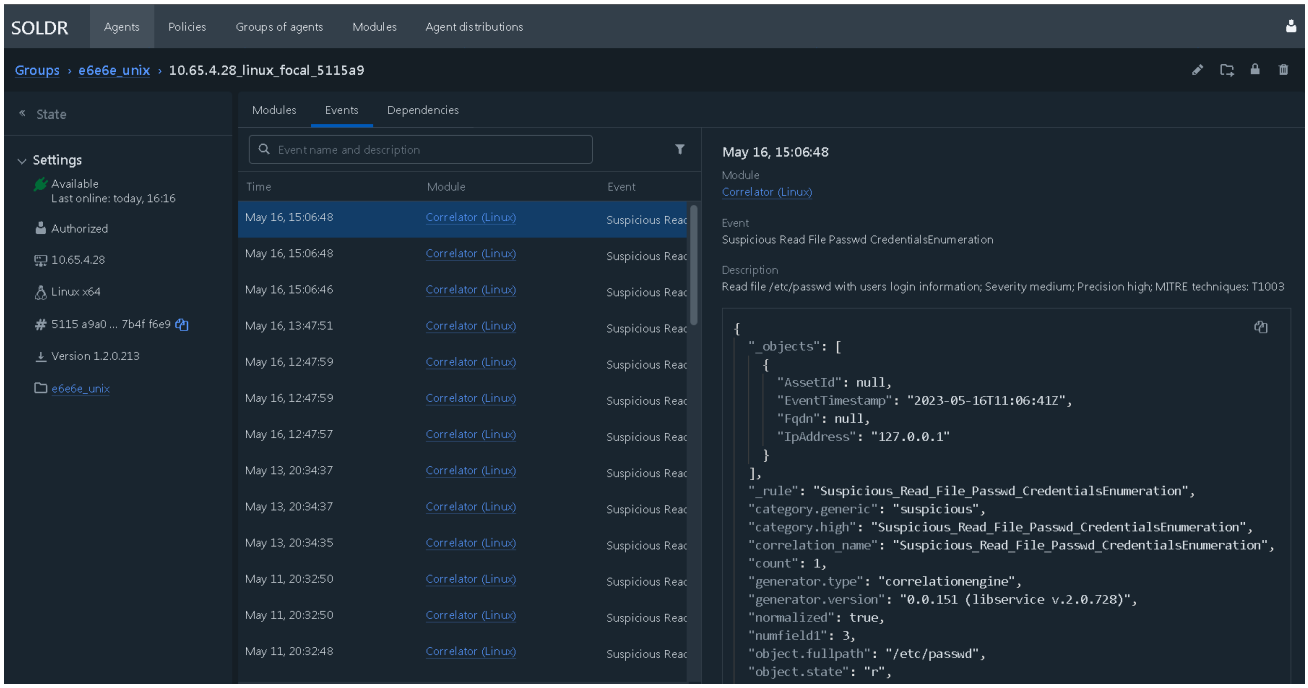
  -a always,exit -F arch=b64 -S connect -F exe=/opt/apache2/bin/httpd -k connect_www

  -a always,exit -F arch=b32 -S listen -F exe=/opt/apache2/bin/httpd -k listen_www

Веб-шелл имеет возможность сетевого взаимодействия. Важно смотреть на сетевые подключения: входящие, исходящие, создание слушающих сокетов.

Поток событий, полученных по этим правилам, запускаем в модуле коррелятора.

На скриншоте alert, что через веб-шелл прочитался файл /etc/passwd.



Обнаружение средствами SELinux

SELinux как известно, может работать в *permissive mode и* enforcing mode. Первый режим только логирует события, не блокируя их. Второй же блокирует запрещенные в применяемых политиках события. Нас интересуют две булевые настройки SELinux:

- allow_execmod – активирует политику безопасности W^X, в соответствии с которой страница памяти одновременно может быть только либо исполняемой, либо доступной для записи. А также запрещает изменение прав страницы памяти с

write (W) на exec (X)

- allow_execmem – запрещает создание страниц памяти с правами RWX

При внедрении кода, в ROP-цепочке изменяются права у страницы памяти. Это успешно детектируется SELinux. Смотрим лог SELinux командой «sudo audit2allow -w -a» и находим там запись:

```
type=AVC msg=audit(1703626677.703:266312): avc:  denied  { execmod } for  pid=3206023 comm="httpd"
 path="/opt/apache2/modules/lib_7.4.33.so" dev="sda5" ino=5636129 scontext=system_u:system_r:initr
c_t:s0 tcontext=system_u:object_r:usr_t:s0 tclass=file permissive=1
        Was caused by:
        The boolean allow_execmod was set incorrectly.
        Description:
        Allow all unconfined executables to use libraries requiring text relocation that are not l
abeled textrel_shlib_t")
```

Остается настроить политики SELinux для httpd и включить enforcing mode.

Общие рекомендации

Самый простой и результативный способ защиты — это вовремя обновлять ПО. Необходимо помнить, что низкоуровневые баги могут быть не только в самом PHP, но и в других используемых им библиотеках, таких как Sqlite3, Libxml2, CURL, libpng и других, которые также требуют своевременного обновления.

Расширения PHP тоже могут содержать низкоуровневые баги. Необходимо отключать в файле конфигурации PHP неиспользуемые в коде веб-приложения расширения. К редко используемым можно отнести, например, расширения GMP и FFI.

Заходите на сайт Cyber OK, чтобы найти еще больше экспертных статей от наших ведущих специалистов и быть на шаг впереди потенциальных угроз. Мы регулярно рассказываем о самых горячих трендах кибербезопасности, чтобы вы оставались в центре событий виртуального мира.