

通过openrasp PHP 源码 查看核心原理

0x01 OpenRASP 介绍

重点阅读部分

而我主要关心的是：

1. agent/php7 (php7版本)
2. agent/php5 (php5版本)

0x02 OpenRASP安装

我这里使用的PHP7.2 编译时带--enable-debug 参数

```
yum install -y centos-release-scl vim-common  
yum install -y devtoolset-7-gcc-c++
```

```
scl enable devtoolset-7 bash
```

安装高版本 `cmake`

```
# 下载并解压到 /tmp, 避免与已有 cmake 冲突  
curl -L https://github.com/Kitware/CMake/releases/download/v3.15.3/cmake-3.15.3-Linux-x86_64.tar.gz | tar zx -C /tmp  
  
# 增加临时 PATH  
export PATH=/tmp/cmake-3.15.3-Linux-x86_64/bin:$PATH
```

编译 openrasp-v8 基础库

在OpenRASP仓库根目录执行以下命令

```
#下载openrasp  
git clone https://github.com/baidu/openrasp.git  
# 更新 git submodule  
git submodule update --init  
  
# 编译 openrasp-v8  
mkdir -p openrasp-v8/build && cd openrasp-v8/build  
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DENABLE_LANGUAGES=php ..  
make
```

编译 OpenRASP PHP 扩展

进入源代码目录，执行下面的命令即可；如果你的 PHP 是自己编译的，请使用对应路径的 `phpize` 命令。

```
# 如果之前编译过，清理下临时文件
/www/server/php/72/bin/phpize --clean

# 生成 configure 文件
/www/server/php/72/bin/phpize

# 生成 makefile
./configure --with-openrasp-v8=../../openrasp-v8/ --with-gettext --enable-
openrasp-remote-manager --enable-debug --with-php-
config=/www/server/php/72/bin/php-config

# 编译
make

#安装
make install

#重启php
/etc/init.d/php-fpm-72 restart
```

然后设置一下配置文件和下载配置文件。参考：<https://rasp.baidu.com/doc/install/manual/php.html>

0x03 OpenRASP PHP7 流程

这里因为使用的是php的插件。首先需要了解一下php插件的几个流程

PHP_GINIT_FUNCTION：全局初始化函数。在每个进程启动时调用，用于初始化全局变量和数据结构。

PHP_GSHUTDOWN_FUNCTION：全局关闭函数。在每个进程关闭时调用，用于清理全局资源。

PHP_INI_BEGIN 和 PHP_INI_END：这些宏定义用于定义 PHP 扩展的配置项。它们不是独立的函数，而是用于包裹配置项的定义。

PHP_MINIT_FUNCTION：模块初始化函数。在 PHP 扩展加载时调用，用于初始化模块相关的功能，例如注册函数、类、常量等。多个扩展的初始化函数的执行顺序可以根据依赖关系和编译顺序确定。

PHP_MSHUTDOWN_FUNCTION：模块关闭函数。在 PHP 扩展卸载时调用，用于清理模块相关的资源。

PHP_RINIT_FUNCTION：请求初始化函数。在每个请求开始时调用，用于初始化请求相关的数据。

PHP_RSHUTDOWN_FUNCTION：请求关闭函数。在每个请求结束时调用，用于清理请求相关的资源。

PHP_MINFO_FUNCTION: 模块信息函数。用于提供模块的版本和相关信息，可以通过 `phpinfo()` 函数来看。

最主要的可以看PHP_MINIT_FUNCTION 和 PHP_RINIT_FUNCTION 这两个。

首先可以看看PHP_MINIT_FUNCTION

```
//初始化和注册模块的功能和资源
PHP_MINIT_FUNCTION(openrasp)
{
    //模块名（openrasp），全局变量初始化函数（PHP_GINIT(openrasp)），以及全局变量清理函数（PHP_GSHUTDOWN(openrasp)）
    ZEND_INIT_MODULE_GLOBALS(openrasp, PHP_GINIT(openrasp),
    PHP_GSHUTDOWN(openrasp));
    //模块卸载时清理全局变量
    REGISTER_INI_ENTRIES();

    //判断SAPI 是否被支持
    if (!current_sapi_supported())
    {
        //如果不支持则就直接退出
        openrasp_status = "Unprotected (unsupported SAPI)";
        return SUCCESS;
    }
    //新建所需要的目录 如果没有权限则退出
    if (!make_openrasp_root_dir(openrasp_ini.root_dir))
    {
        openrasp_status = "Unprotected ('openrasp.root_dir' initialization failed)";
        return SUCCESS;
    }
    //root_dir/locale/ 目录
    std::string locale_path(std::string(openrasp_ini.root_dir) + DEFAULT_SLASH + "locale" + DEFAULT_SLASH);
    //配置的语言类型
    openrasp_set_locale(openrasp_ini.locale, locale_path.c_str());
    //RASP ID 查看是否符合规定
    if (!openrasp_ini.verify_rasp_id())
    {
        openrasp_error(LEVEL_WARNING, CONFIG_ERROR, _("openrasp.rasp_id can only contain alphanumeric characters and is between 16 and 512 in length.));
        openrasp_status = "openrasp.rasp_id can only contain alphanumeric characters and is between 16 and 512 in length.";
        return SUCCESS;
    }
    //
    openrasp::scm.reset(new openrasp::SharedConfigManager());
    //共享内存中创建一个配置块，并初始化相关的成员变量和数据。它会创建一个读写锁对象
    if (!openrasp::scm->startup())
    {
        openrasp_error(LEVEL_WARNING, RUNTIME_ERROR, _("Fail to startup SharedConfigManager.));
    }
}
```

```

        openrasp_status = "Unprotected (shared memory application failed)";
        return SUCCESS;
    }

#ifdef HAVE_OPENRASP_REMOTE_MANAGER
    //判断是否是php 的sapi 方式和 是否开启远程管理功能
    if (need_alloc_shm_current_sapi() && openrasp_ini.remote_management_enable)
    {
        //如果开启远程管理的情况下初始化一个管理的对象
        openrasp::oam.reset(new openrasp::OpenraspAgentManager());
        std::string error_msg;
        //判断一下ini的信息是否OK
        if (!openrasp_ini.verify_remote_management_ini(error_msg))
        {
            openrasp_status = error_msg;
            openrasp_error(LEVEL_WARNING, CONFIG_ERROR, error_msg.c_str());
            return SUCCESS;
        }
        remote_active = true;
    }
#endif
    //于初始化 openrasp_log 模块
    if (PHP_MINIT(openrasp_log)(INIT_FUNC_ARGS_PASSTHRU) == FAILURE)
    {
        openrasp_status = "Unprotected (log module initialization failed)";
        return SUCCESS;
    }
    //即远程管理功能未激活
    if (!remote_active)
    {
        openrasp::YamlReader
        yaml_reader(get_complete_config_content(ConfigHolder::FromType::kYaml));
        yaml_reader.set_exception_report(true);
        if (yaml_reader.has_error())
        {
            openrasp_error(LEVEL_WARNING, CONFIG_ERROR, _("Fail to parse config,
            cuz of %s."),
                        yaml_reader.get_error_msg().c_str());
        }
        else
        {
            std::string unknown_yaml_keys =
            yaml_reader.detect_unknown_config_key();
            if (!unknown_yaml_keys.empty())
            {
                openrasp_error(LEVEL_WARNING, CONFIG_ERROR, _("Unknown config key
                [%s] found in yaml configuration."),
                            unknown_yaml_keys.c_str());
            }
            openrasp::scm->set_debug_level(&yaml_reader);
            openrasp::scm->build_check_type_white_array(&yaml_reader);
            openrasp::scm->build_weak_password_array(&yaml_reader);
            OPENRASP_G(config).update(&yaml_reader);
        }
    }
}
//判断v8 模块是否正常

```

```

if (PHP_MINIT(openrasp_v8)(INIT_FUNC_ARGS_PASSTHRU) == FAILURE)
{
    openrasp_status = "Unprotected (v8 module initialization failed)";
    return SUCCESS;
}
int result;
//初始化 hook 和inject
result = PHP_MINIT(openrasp_hook)(INIT_FUNC_ARGS_PASSTHRU);
result = PHP_MINIT(openrasp_inject)(INIT_FUNC_ARGS_PASSTHRU);

#ifdef HAVE_OPENRASP_REMOTE_MANAGER
//如果开启了远程管理功能 并且oam 也有对象
if (remote_active && openrasp::oam)
{
    //如果name 为cgi-fcgi 时候 忽略子进程的终止信号
    if (sapi_module.name && strcmp(sapi_module.name, "cgi-fcgi") == 0)
    {
        signal(SIGCHLD, SIG_IGN);
    }
    //启动 OpenRASP 的远程管理功
    openrasp::oam->startup();
}
#endif
#ifdef HAVE_FSWATCH
//根据远程管理功能的状态选择性地初始化文件系统监视功能
if (!remote_active)
{
    result = PHP_MINIT(openrasp_fswatch)(INIT_FUNC_ARGS_PASSTHRU);
}
#endif
//初始化openrasp_security_policy
result = PHP_MINIT(openrasp_security_policy)(INIT_FUNC_ARGS_PASSTHRU);
//初始化openrasp_output_detect
result = PHP_MINIT(openrasp_output_detect)(INIT_FUNC_ARGS_PASSTHRU);
is_initialized = true;
return SUCCESS;
}

```

核心点:

```

if (PHP_MINIT(openrasp_v8)(INIT_FUNC_ARGS_PASSTHRU) == FAILURE)
{
    openrasp_status = "Unprotected (v8 module initialization failed)";
    return SUCCESS;
}
int result;
//初始化 hook 和inject
result = PHP_MINIT(openrasp_hook)(INIT_FUNC_ARGS_PASSTHRU);

```

初始化openrasp_v8 模块。为后续的js引擎做准备。

我们看看官网的介绍

OpenRASP 核心原理为：在 MINIT 阶段，替换全局 `compiler_globals` 的 `function_table` 与 `class_table` 中特定 `PHP_FUNCTION` 对应的函数指针（封装原有 `handler`，增加前置、后置处理），由此实现对敏感函数的挂钩。通过敏感函数参数结合请求信息判断是否存在攻击行为，进而采取拦截或者放行操作。

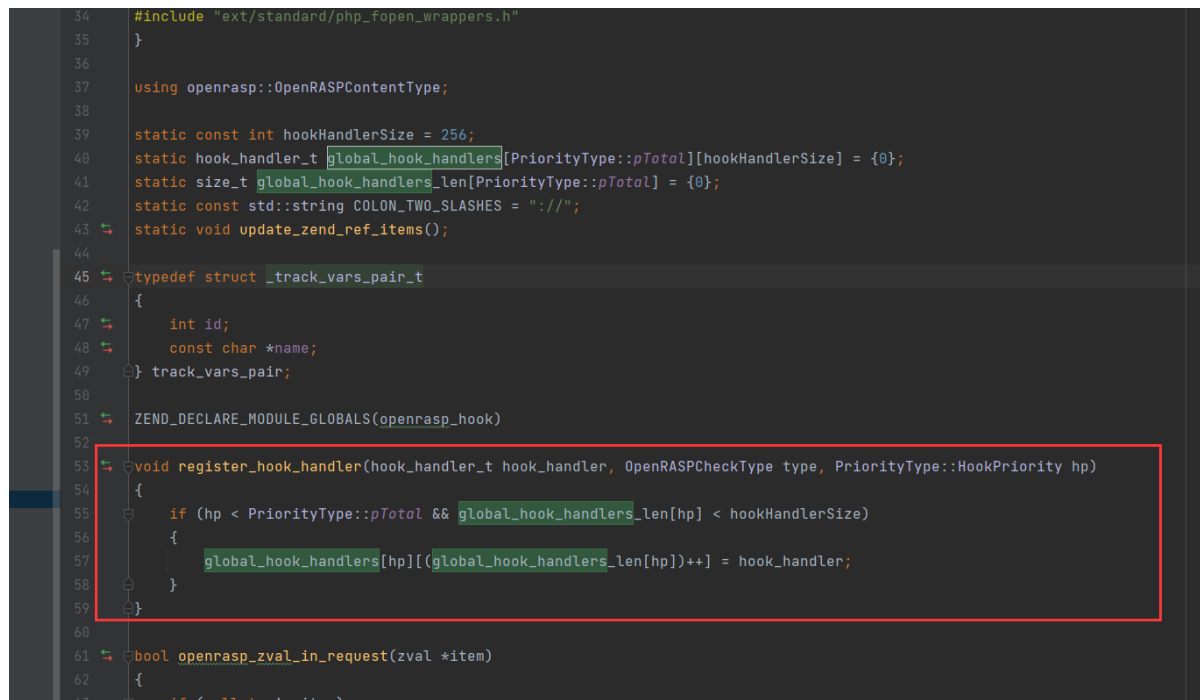
看看他是怎么实现的。

首先找到 `PHP_MINIT(openrasp_hook)(INIT_FUNC_ARGS_PASSTHRU)` 中他怎么 hook 的

```
PHP_MINIT_FUNCTION(openrasp_hook)
{
    ZEND_INIT_MODULE_GLOBALS(openrasp_hook, PHP_GINIT(openrasp_hook),
    PHP_GSHUTDOWN(openrasp_hook));
    for (size_t i = 0; i < PriorityType::pTotal; ++i)
    {
        for (size_t j = 0; j < global_hook_handlers_len[i]; ++j)
        {
            global_hook_handlers[i][j]();
        }
    }
    zend_set_user_opcode_handler(ZEND_INCLUDE_OR_EVAL, include_or_eval_handler);
    zend_set_user_opcode_handler(ZEND_ECHO, echo_print_handler);
    return SUCCESS;
}
```

这里只看到了 `include` 和 `echo` 的 hook 并没有看到有其他的 hook 点。但是很奇怪的是执行了一个 `global_hook_handlers[i][j]();` 这个函数

首先看看 `global_hook_handlers` 到底是怎么来的



```
34 #include "ext/standard/php_fopen_wrappers.h"
35 }
36
37 using openrasp::OpenRASPCheckType;
38
39 static const int hookHandlerSize = 256;
40 static hook_handler_t global_hook_handlers[PriorityType::pTotal][hookHandlerSize] = {0};
41 static size_t global_hook_handlers_len[PriorityType::pTotal] = {0};
42 static const std::string COLON_TWO_SLASHES = "://";
43 static void update_zend_ref_items();
44
45 typedef struct _track_vars_pair_t
46 {
47     int id;
48     const char *name;
49 } track_vars_pair;
50
51 ZEND_DECLARE_MODULE_GLOBALS(openrasp_hook)
52
53 void register_hook_handler(hook_handler_t hook_handler, OpenRASPCheckType type, PriorityType::HookPriority hp)
54 {
55     if (hp < PriorityType::pTotal && global_hook_handlers_len[hp] < hookHandlerSize)
56     {
57         global_hook_handlers[hp][(global_hook_handlers_len[hp])++] = hook_handler;
58     }
59 }
60
61 bool openrasp_zval_in_request(zval *item)
62 {
63     if (evalstate.is_item)
```

搜索了一圈。只有一个地方调用了 `openrasp_hook.h` 中的 `OPENRASP_HOOK_FUNCTION_PRIORITY_EX` 宏调用他了。

那么看看这个宏又是被谁调用的

```

238 #define HOOK_FUNCTION(name, type) \
239     HOOK_FUNCTION_PRIORITY(name, type, PriorityType::pNormal)
240
241 //前置hook
242 #define PRE_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority) \
243     void pre_##scope##_##name##_##type(OPENRASP_INTERNAL_FUNCTION_PARAMETERS); \
244     OPENRASP_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority) \
245     { \
246         bool type_ignored = openrasp_check_type_ignored(type); \
247         if (!type_ignored) \
248         { \
249             pre_##scope##_##name##_##type(INTERNAL_FUNCTION_PARAM_PASSTHRU, (type)); \
250         } \
251         origin_function(INTERNAL_FUNCTION_PARAM_PASSTHRU); \
252     }
253
254 #define PRE_HOOK_FUNCTION_EX(name, scope, type) \
255     PRE_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, PriorityType::pNormal)
256
257 #define PRE_HOOK_FUNCTION_PRIORITY(name, type, priority) \
258     PRE_HOOK_FUNCTION_PRIORITY_EX(name, global, type, priority)
259
260 #define PRE_HOOK_FUNCTION(name, type) \
261     PRE_HOOK_FUNCTION_PRIORITY(name, type, priority: PriorityType::pNormal)
262
263 //后置hook
264 #define POST_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority) \
265     void post_##scope##_##name##_##type(OPENRASP_INTERNAL_FUNCTION_PARAMETERS); \
266     OPENRASP_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority) \
267     { \
268         origin_function(INTERNAL_FUNCTION_PARAM_PASSTHRU); \
269         bool type_ignored = openrasp_check_type_ignored(type); \
270         if (!type_ignored) \
271         { \
272             post_##scope##_##name##_##type(INTERNAL_FUNCTION_PARAM_PASSTHRU, (type)); \
273         } \
274     }
275
276 #define POST_HOOK_FUNCTION_EX(name, scope, type) \
277     POST_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, PriorityType::pNormal)

```

发现这个宏最终能调用的为: <https://rasp.baidu.com/doc/hacking/hook.html>

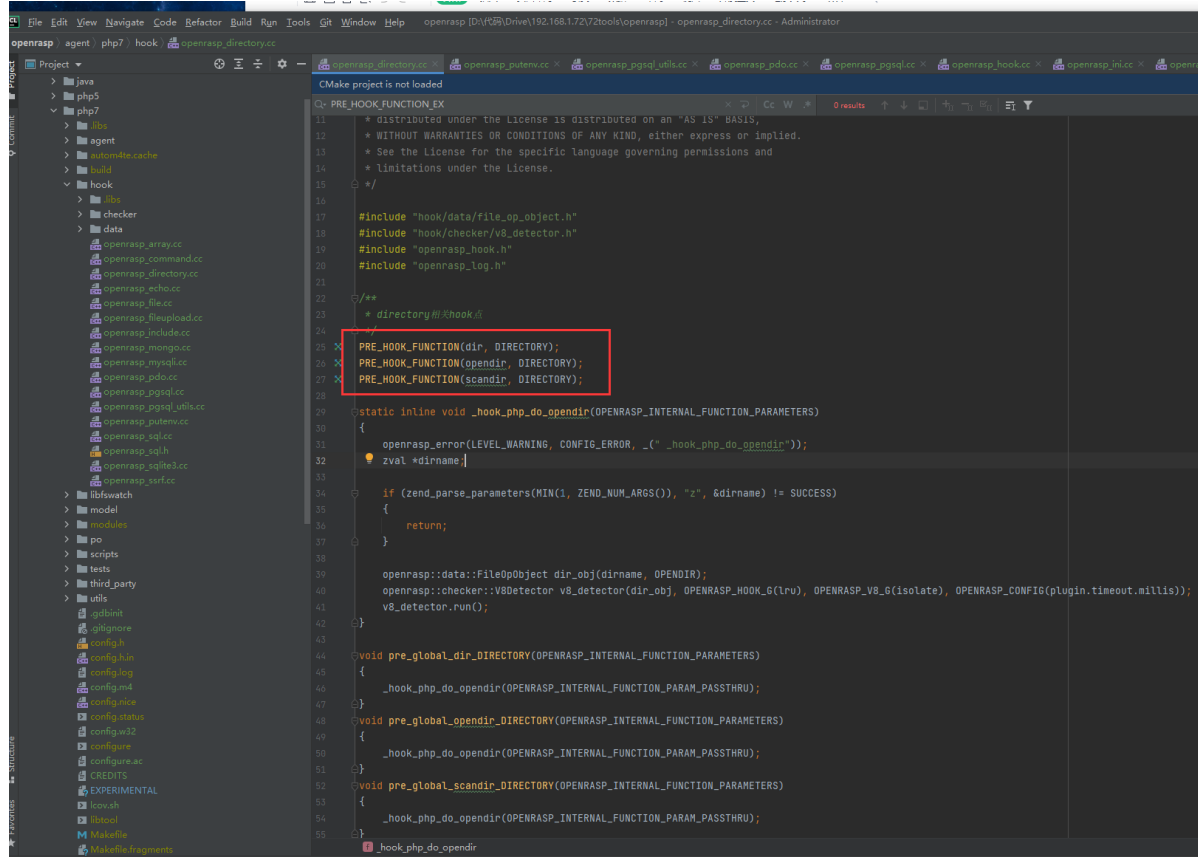
官网描述的三个宏:

前置HOOK(PRE_HOOK_FUNCTION_EX),

后置HOOK(POST_HOOK_FUNCTION_EX)

前后同时HOOK(HOOK_FUNCTION_EX)

那么我们搜索一下任意一个。看看它是一个怎样子的调用过程



随便找了一个hook 的地方。看看他整一个调用过程

```
#define PRE_HOOK_FUNCTION(name, type) \  
    PRE_HOOK_FUNCTION_PRIORITY(name, type, PriorityType::pNormal)  
#define PRE_HOOK_FUNCTION_PRIORITY(name, type, priority) \  
    PRE_HOOK_FUNCTION_PRIORITY_EX(name, global, type, priority)
```

使用了PRE_HOOK_FUNCTION_PRIORITY_EX 的宏。

通过定义一个pre##scope####name##_##type 在函数之前执行。


```
//前置hook
#define PRE_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority)
\
void pre_##scope##_##name##_##type(OPENRASP_INTERNAL_FUNCTION_PARAMETERS);
\
OPENRASP_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority)
\
{
\
bool type_ignored = openrasp_check_type_ignored(type);
\
if (!type_ignored)
\
{
\
pre_##scope##_##name##_##type(INTERNAL_FUNCTION_PARAM_PASSTHRU,
(type)); \
}
\
origin_function(INTERNAL_FUNCTION_PARAM_PASSTHRU);
\
}
```

OPENRASP_HOOK_FUNCTION_PRIORITY_EX 宏的核心。是通过DEFINE_HOOK_HANDLER_EX hook到 name 的function_table 然后使用register_hook_handler 函数进行一个回调。

```
#define OPENRASP_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority)
php_function origin_##scope##_##name##_##type = nullptr;

inline void hook_##scope##_##name##_##type##_ex(INTERNAL_FUNCTION_PARAMETERS,
php_function origin_function);
void hook_##scope##_##name##_##type(INTERNAL_FUNCTION_PARAMETERS)

{

hook_##scope##_##name##_##type##_ex(INTERNAL_FUNCTION_PARAM_PASSTHRU,
origin_##scope##_##name##_##type); \
}

DEFINE_HOOK_HANDLER_EX(name, scope, type);

int scope##_##name##_##type = []() {

register_hook_handler(scope##_##name##_##type##_handler, type,
priority);return 0; }();

inline void hook_##scope##_##name##_##type##_ex(INTERNAL_FUNCTION_PARAMETERS,
php_function origin_function)
```

DEFINE_HOOK_HANDLER_EX 实现了。函数hook

```

#define DEFINE_HOOK_HANDLER_EX(name, scope, type)

void scope##_##name##_##type##_handler()

{

    HashTable *ht = nullptr;

    zend_function *function;

    if (strcmp("global", ZEND_TOSTR(scope)) == 0)

    {

        ht = CG(function_table);

    }

    else

    {

        zend_class_entry *clazz;

        std::string scope_str(ZEND_TOSTR(scope));
        openrasp::string_replace(scope_str, ZEND_TOSTR(BACKSLASH_IN_CLASS),
"\\");

        if ((clazz = static_cast<zend_class_entry *>(

                                zend_hash_str_find_ptr(CG(class_table), scope_str.c_str(),
scope_str.length())) != NULL)

            {

                ht = &(clazz->function_table);

            }

        }

        if (ht && (function = static_cast<zend_function *>
(zend_hash_str_find_ptr(ht, ZEND_STRL(ZEND_TOSTR(name))))) != NULL &&
            function->internal_function.handler != zif_display_disabled_function)

        {

            origin_##scope##_##name##_##type = function-
>internal_function.handler;
            function->internal_function.handler = hook_##scope##_##name##_##type;

        }

    }

}

```

然后每个hook 函数都需要按照函数的命名的方式来进行hook 例如：

```
PRE_HOOK_FUNCTION(dir, DIRECTORY);
PRE_HOOK_FUNCTION(opendir, DIRECTORY);
PRE_HOOK_FUNCTION(scandir, DIRECTORY);
```

对应的函数如下:

```
void pre_global_dir_DIRECTORY(OPENRASP_INTERNAL_FUNCTION_PARAMETERS)
{
    _hook_php_do_opendir(OPENRASP_INTERNAL_FUNCTION_PARAM_PASSTHRU);
}
void pre_global_opendir_DIRECTORY(OPENRASP_INTERNAL_FUNCTION_PARAMETERS)
{
    _hook_php_do_opendir(OPENRASP_INTERNAL_FUNCTION_PARAM_PASSTHRU);
}
void pre_global_scandir_DIRECTORY(OPENRASP_INTERNAL_FUNCTION_PARAMETERS)
{
    _hook_php_do_opendir(OPENRASP_INTERNAL_FUNCTION_PARAM_PASSTHRU);
}
```

前置钩子和后置的钩子的区别在于。如下：一个是在
origin_function(INTERNAL_FUNCTION_PARAM_PASSTHRU); 执行后执行的。一个是在执行前执行的。

```
#define POST_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority)
\
void post_##scope##_##name##_##type(OPENRASP_INTERNAL_FUNCTION_PARAMETERS);
\
OPENRASP_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority)
\
{
\
    origin_function(INTERNAL_FUNCTION_PARAM_PASSTHRU);
\
    bool type_ignored = openrasp_check_type_ignored(type);
\
    if (!type_ignored)
\
    {
\
        post_##scope##_##name##_##type(INTERNAL_FUNCTION_PARAM_PASSTHRU,
(type)); \
    }
\
}
```

还有一个是两个都hook的 前执行pre 的函数。后执行post的函数

```
#define HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority)
\
void pre_##scope##_##name##_##type(OPENRASP_INTERNAL_FUNCTION_PARAMETERS);
\
```

```

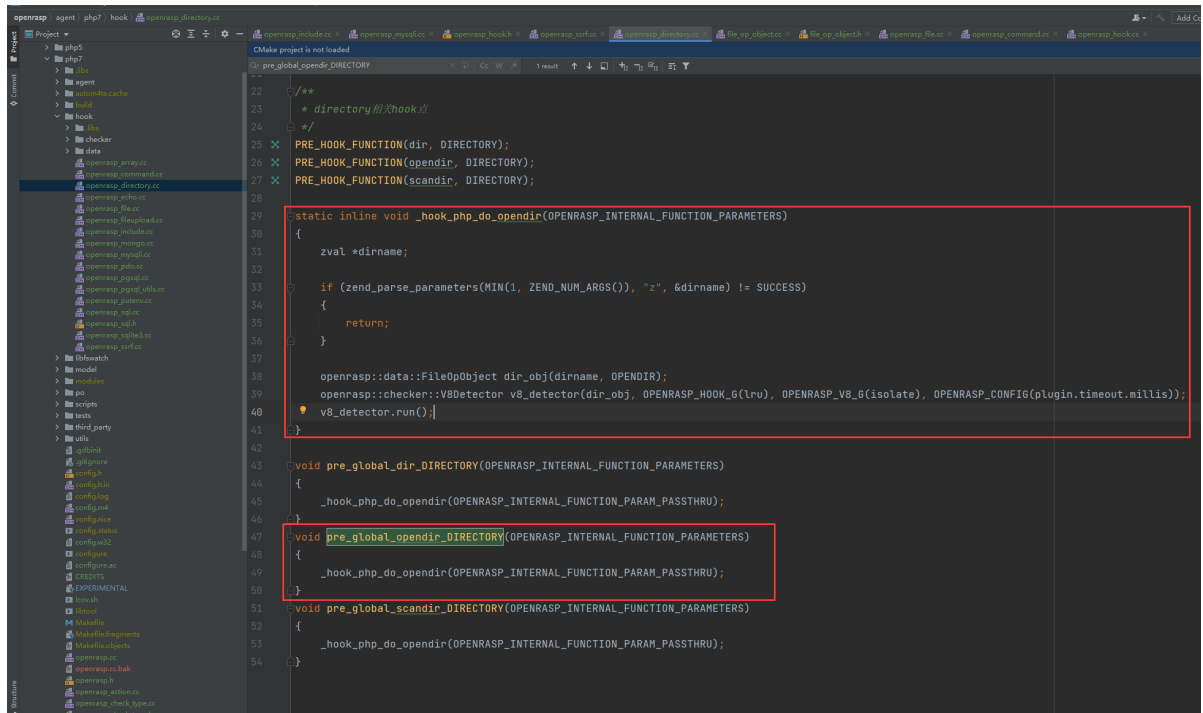
void post_##scope##_##name##_##type(OPENRASP_INTERNAL_FUNCTION_PARAMETERS);
\
OPENRASP_HOOK_FUNCTION_PRIORITY_EX(name, scope, type, priority)
\
{
\
    bool pre_type_ignored = openrasp_check_type_ignored(type);
\
    if (!pre_type_ignored)
\
    {
\
        pre_##scope##_##name##_##type(INTERNAL_FUNCTION_PARAM_PASSTHRU,
(type)); \
    }
\
    origin_function(INTERNAL_FUNCTION_PARAM_PASSTHRU);
\
    bool post_type_ignored = openrasp_check_type_ignored(type);
\
    if (!post_type_ignored)
\
    {
\
        post_##scope##_##name##_##type(INTERNAL_FUNCTION_PARAM_PASSTHRU,
(type)); \
    }
\
}

```

这大概就是openrasp 二次开发中所说的原理

0x04 OpenRASP 拦截的实现

例如：opendir



前置hook

```
void pre_global_opendir_DIRECTORY(OPENRASP_INTERNAL_FUNCTION_PARAMETERS)
{
    _hook_php_do_opendir(OPENRASP_INTERNAL_FUNCTION_PARAM_PASSTHRU);
}
```

最终到达:

```
static inline void _hook_php_do_opendir(OPENRASP_INTERNAL_FUNCTION_PARAMETERS)
{
    zval *dirname;
    if (zend_parse_parameters(MIN(1, ZEND_NUM_ARGS()), "z", &dirname) != SUCCESS)
    {
        return;
    }
    openrasp::data::FileOpObject dir_obj(dirname, OPENDIR);
    openrasp::checker::V8Detector v8_detector(dir_obj, OPENRASP_HOOK_G(lru),
    OPENRASP_V8_G(isolate), OPENRASP_CONFIG(plugin.timeout.millis));
    v8_detector.run();
}
```

openrasp::data::FileOpObject 这个对象是V8Material的子对象。所有需要进行拦截的类型都需要是V8Material的子对象

```

16
17     #pragma once
18
19     #include "openrasp_hook.h"
20     #include "php_openrasp.h"
21     #include "v8_material.h"
22
23     namespace openrasp
24     {
25     namespace data
26     {
27
28     class FileOpObject : public V8Material
29     {
30     private:
31         //do not efree here
32         zval *file = nullptr;
33         bool use_include_path = false;
34         PathOperation w_op;
35         std::string realpath;
36
37     public:
38         FileOpObject(zval *file, PathOperation w_op, bool use_include_path = false);
39         virtual bool is_valid() const;
40
41         //v8
42         virtual std::string build_lru_key() const;
43         virtual OpenRASPCheckType get_v8_check_type() const;
44         virtual void fill_object_2b_checked(Isolate *isolate, v8::Local<v8::Object> params) const;
45     };
46
47     // namespace data
48
49     // namespace openrasp

```

实例化完FileOpObject 后 把对象传递给V8Detector 进行实例化

最终使用run函数进行判断。

V8Detector

```

V8Detector::V8Detector(const openrasp::data::V8Material &v8_material,
openrasp::LRU<std::string, bool> &lru, openrasp::Isolate *isolate, int timeout,
bool canBlock)
    : v8_material(v8_material), lru(lru), isolate(isolate), timeout(timeout),
canBlock(canBlock)
{
}

```

run 的逻辑大概如下:

```

void V8Detector::run()
{
    if (!pretreat())
    {
        return;
    }

    std::string lru_ley = v8_material.build_lru_key();
    if (!lru_ley.empty() &&
        lru.contains(lru_ley))
    {
        return;
    }
}

```

```

    CheckResult cr = check();
    if (kNoCache == cr)
    {
        return;
    }
    else if (kCache == cr)
    {
        lru.set(lru_ley, true);
    }
    else if (kBlock == cr && canBlock)
    {
        //终止当前请求的执行
        block_handle();
    }
}

```

通过 pretreat 判断是否是为空的。为空就不处理。

通过 build_lru_key 获取的 LRU 键值 如果存在就直接返回

在执行 check 前。会把他的参数设置到 V8 的上下文当中

```

//它用于在给定的 v8 上下文中填充一个对象
//获取 v8 上下文，设置两个属性 path realpath
void FileOpObject::fill_object_2b_checked(Isolate *isolate, v8::Local<v8::Object>
params) const
{
    v8::HandleScope handle_scope(isolate);
    auto context = isolate->GetCurrentContext();
    params->Set(context, openrasp::NewV8String(isolate, "path"),
openrasp::NewV8String(isolate, Z_STRVAL_P(file), Z_STRLEN_P(file))).IsJust();
    params->Set(context, openrasp::NewV8String(isolate, "realpath"),
openrasp::NewV8String(isolate, realpath)).IsJust();
}

```

check 为实际的检测方法。如果为 kBlock 则 终止当前的请求。

然后怎么交互的看看 js 怎么实现的这个过滤

```

plugin.register('directory', function (params, context) {

    var realpath    = params.realpath
    var server      = context.server

    var is_windows  = server.os.indexOf('windows') != -1
    var language    = server.language

    // 算法2 - 检查PHP菜刀等后门
    if (algorithmConfig.directory_reflect.action != 'ignore')
    {
        if (language == 'php' && validate_stack_php(params.stack))
        {
            return {
                action:    algorithmConfig.directory_reflect.action,

```

```

        message:    _("webShell activity - Using file manager function
with China Chopper webShell"),
        confidence: 90,
        algorithm:  'directory_reflect'
    }
}
else if (language == 'java' && validate_stack_java(params.stack) &&
!is_method_from_rasp(params.stack))
{
    return {
        action:      algorithmConfig.directory_reflect.action,
        message:      _("webShell activity - Using file manager function
with Java webShell"),
        confidence: 90,
        algorithm:    'directory_reflect'
    }
}
}

// 算法1 - 用户输入匹配。
if (algorithmConfig.directory_userinput.action != 'ignore')
{
    var all_parameter = get_all_parameter(context)

    if (is_path_containing_userinput(all_parameter, params.path, is_windows,
algorithmConfig.directory_userinput.lcs_search))
    {
        return {
            action:      algorithmConfig.directory_userinput.action,
            message:      _("Path traversal - Accessing folder specified by
userinput, folder is %1%", [realpath]),
            confidence: 90,
            algorithm:    'directory_userinput'
        }
    }
}

// 算法3 - 读取敏感目录
if (algorithmConfig.directory_unwanted.action != 'ignore')
{
    for (var i = 0; i < forcefulBrowsing.unwantedDirectory.length; i++) {
        if (realpath == forcefulBrowsing.unwantedDirectory[i]) {
            return {
                action:      algorithmConfig.directory_unwanted.action,
                message:      _("webShell activity - Accessing sensitive
folder: %1%", [realpath]),
                confidence: 100,
                algorithm:    'directory_unwanted'
            }
        }
    }
}

return clean
})

```


首先检查用户输入的内容和。检查敏感目录

```
unwantedDirectory: [  
    '/',  
    '/home',  
    '/var/log',  
    '/private/var/log',  
    '/proc',  
    '/sys',  
    'c:\\',  
    'd:\\',  
    'E:\\'  
],
```

然后他的算法是记录日志

```
// 文件管理器 - 查看敏感目录  
directory_unwanted: {  
    name: '算法3 - 尝试查看敏感目录',  
    action: 'log'  
},
```

使用官网的案例

← → ↻ ▲ 不安全 | 192.168.1.72/vulns/001-dir.php?dir=/home

001 - 列目录操作 - scandir 方式

若测试用例无法执行，请检查 open_basedir 配置，以及目录是否有读取权限。

Linux 不正常调用:

```
curl -g 'http://192.168.1.72/vulns/001-dir.php?dir=../../../../../../../../etc'
```

Linux 不正常调用 (json方式) :

```
curl -d '{"dir": "../../../../../../../../etc"}' -H "Content-Type: application/json" 'http://192.168.1.72/vulns/001-dir.php'  
click here to access ../../../../../../etc/
```

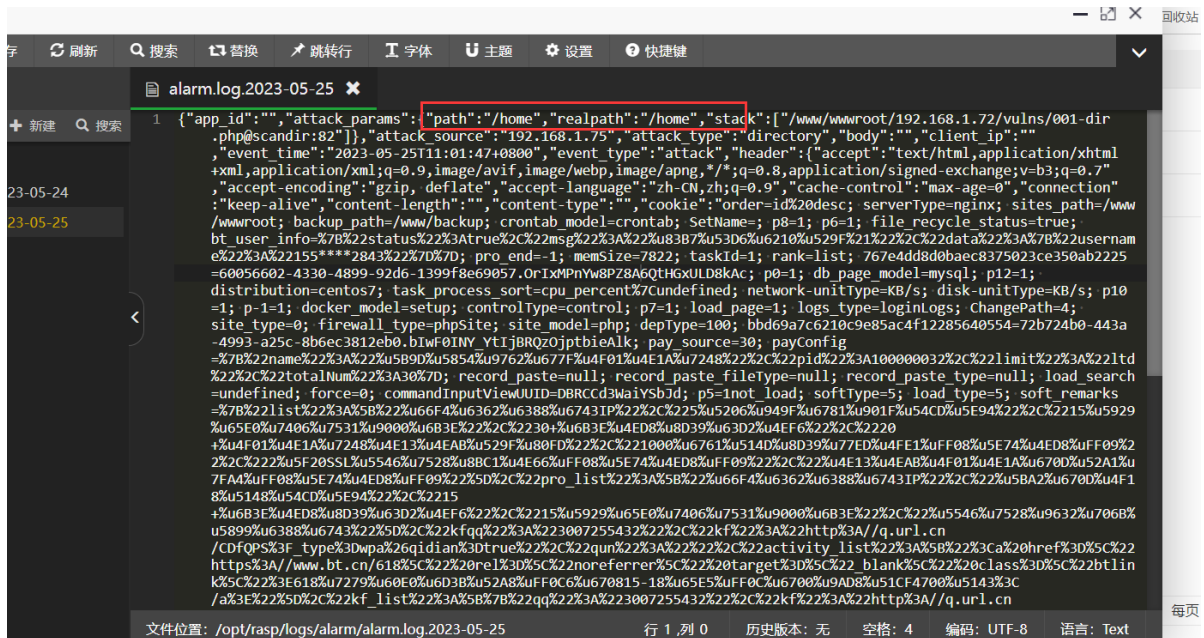
windows 不正常调用:

```
curl -g 'http://192.168.1.72/vulns/001-dir.php?dir=../../../../../../../../windows'
```

目录内容

```
.  
..  
redis  
test  
www
```

查看一下日志



0x04 OpenRASP 其他

V8这块确实没怎么看懂， IAST 没有去看。这里只是看了部分代码。大概百分之10-15左右吧