Python中从服务端模板注入到沙盒逃逸的源码探索 (一)

virink / 2018-10-16 03:02:49 / 浏览数 3986

原理

~~顾名思义，服务端模板注入，就是通过在服务端的模板文件或模板字符串中注入特定的恶意代码导致产生代码执行的一种漏洞攻击方式。~~

不同的模板引擎，根据不同的解析方式相应的也是存在不同的利用方法。

正常而言，出于安全考虑，模板引擎基本上都是拥有沙盒、命名空间的，代码的解析执行都是发生在有限的沙盒里面，因此，沙盒逃逸也成为 SSTI 不可或缺的存在。

Python Web 模板引擎

- [x] Jinja2
- [x] Tornado.template
- [ ] Django.template
- ...

SSTI in Tornado

Tornado 中模板渲染函数在有两个

- render
- render_string

tornado/web.py:

```
class RequestHandler(object):
    ....
    def render(self, template_name, **kwargs):
        ...
        html = self.render_string(template_name, **kwargs)
        ...
        return self.finish(html)

    def render_string(self, template_name, **kwargs):

        template_path = self.get_template_path()
        ...
        with RequestHandler._template_loader_lock:
            if template_path not in RequestHandler._template_loaders:
                loader = self.create_template_loader(template_path)
                RequestHandler._template_loaders[template_path] = loader
            else:
                loader = RequestHandler._template_loaders[template_path]
        t = loader.load(template_name)
        namespace = self.get_template_namespace()
        namespace.update(kwargs)
        return t.generate(**namespace)

    def get_template_namespace(self):
        namespace = dict(
            handler=self,
            request=self.request,
            current_user=self.current_user,
            locale=self.locale,
            _=self.locale.translate,
            pgettext=self.locale.pgettext,
            static_url=self.static_url,
            xsrf_form_html=self.xsrf_form_html,
            reverse_url=self.reverse_url
        )
        namespace.update(self.ui)
        return namespace
```

render_string:通过模板文件名加载模板，然后更新模板引擎中的命名空间，添加一些全局函数或其他对象，然后生成并返回渲染好的 html内容

render:依次调用`render_string`及相关渲染函数生成的内容，最后调用 finish 直接输出给客户端。

我们跟进模板引擎相关类看看其中的实现。

tornado/template.py

```
class Template(object):
   ...
   def generate(self, **kwargs):
       namespace = {
           "escape": escape.xhtml_escape,
           "xhtml_escape": escape.xhtml_escape,
           "url_escape": escape.url_escape,
           "json_encode": escape.json_encode,
           "squeeze": escape.squeeze,
           "linkify": escape.linkify,
           "datetime": datetime,
           "_tt_utf8": escape.utf8,  # for internal use
           "_tt_string_types": (unicode_type, bytes),
           "__name__": self.name.replace('.', '_'),
           "__loader__": ObjectDict(get_source=lambda name: self.code),
       }
       namespace.update(self.namespace)
       namespace.update(kwargs)
       exec_in(self.compiled, namespace)
       execute = namespace["_tt_execute"]
       linecache.clearcache()
       return execute()
```

在上面的代码中，我们很容易看出命名空间namespace中有哪些变量、函数的存在。其中，handler是一个神奇的存在。

tornado/web.py:

```
class RequestHandler(object):
   ....
   def __init__(self, application, request, **kwargs):
       super(RequestHandler, self).__init__()

       self.application = application
       self.request = request
```

在`RequestHandler`类的构造函数中，可以看到`application`的赋值。

tornado/web.py:

```
class Application(ReversibleRouter):
   ....
   def __init__(self, handlers=None, default_host=None, transforms=None,
                **settings):
       ...
       self.wildcard_router = _ApplicationRouter(self, handlers)
       self.default_router = _ApplicationRouter(self, [
           Rule(AnyMatches(), self.wildcard_router)
       ])

class _ApplicationRouter(ReversibleRuleRouter):
   def __init__(self, application, rules=None):
       assert isinstance(application, Application)
       self.application = application
       super(_ApplicationRouter, self).__init__(rules)
```

因此，通过`handler.application`即可访问整个Tornado。

简单而言通过`{{handler.application.settings}}`或者`{{handler.settings}}`就可获得`settings`中的cookie_secret。

例题：[护网杯 2018 WEB (1) easy_tornado]

另外，跟进`exec_in`中也有新发现。

tornado/util.py:

```
def exec_in(code, glob, loc=None):
   # type: (Any, Dict[str, Any], Optional[Mapping[str, Any]]) -> Any
   if isinstance(code, basestring_type):
       # exec(string) inherits the caller's future imports; compile
       # the string first to prevent that.
       code = compile(code, '<string>', 'exec', dont_inherit=True)
   exec(code, glob, loc)
```

这里用到了compile和exec

SSTI in Flask

Flask 中模板渲染函数也是有两个

- render_template
- render_template_string

Flask使用的是 Jinja2 模板引擎

flask/templating.py

```
def _default_template_ctx_processor():
   """Injects `request`,`session` and `g`."""
   reqctx = _request_ctx_stack.top
   appctx = _app_ctx_stack.top
   rv = {}
   if appctx is not None:
       rv['g'] = appctx.g
   if reqctx is not None:
       rv['request'] = reqctx.request
       rv['session'] = reqctx.session
   return rv

def _render(template, context, app):
   before_render_template.send(app, template=template, context=context)
   rv = template.render(context)
   template_rendered.send(app, template=template, context=context)
   return rv

def render_template(template_name_or_list, **context):
   ctx = _app_ctx_stack.top
   ctx.app.update_template_context(context)
   return _render(ctx.app.jinja_env.get_or_select_template(template_name_or_list),
                  context, ctx.app)

def render_template_string(source, **context):
   ctx = _app_ctx_stack.top
   ctx.app.update_template_context(context)
   return _render(ctx.app.jinja_env.from_string(source),
                  context, ctx.app)
```

render_template:通过模板文件加载内容并进行渲染

render_template_string:直接通过模板字符串进行渲染

这上下文、栈啥的看的有点懵，也不深入了。（有兴趣自行了解）

接着，我们看看 Flask 是怎么加载 Jinja2 的。`app.jinja_env`

flask/app.py

```
class Flask(_PackageBoundObject):
   ...
   jinja_environment = Environment
   ...
   jinja_options = ImmutableDict(
       extensions=['jinja2.ext.autoescape', 'jinja2.ext.with_']
   )
   ...
```

```
    @locked_cached_property
    def jinja_env(self):
        return self.create_jinja_environment()
    ...
    def create_jinja_environment(self):
        options = dict(self.jinja_options)
        rv = self.jinja_environment(self, **options)
        rv.globals.update(
            url_for=url_for,
            get_flashed_messages=get_flashed_messages,
            config=self.config,
            request=request,
            session=session,
            g=g
        )
        rv.filters['tojson'] = json.tojson_filter
        return rv
```

这里我们可以看见jinja_environment中有6个全局变量，也就是说在模板引擎的解析环境中可以访问这6个对象。

例题 TokyoWesterns CTF 4th 2018 shrine

接下来，我们跟进 Jinja2 的代码里看看还有什么有意思的东西。

jinja2/environment.py

```
class Environment(object):
    ...
    def _generate(self, source, name, filename, defer_init=False):
        return generate(source, self, name, filename, defer_init=defer_init,
                        optimized=self.optimized)

    def _compile(self, source, filename):
        return compile(source, filename, 'exec')

    @internalcode
    def compile(self, source, name=None, filename=None, raw=False,
                defer_init=False):
        source_hint = None
        try:
            if isinstance(source, string_types):
                source_hint = source
                source = self._parse(source, name, filename)
            source = self._generate(source, name, filename,
                                    defer_init=defer_init)
            if raw:
                return source
            if filename is None:
                filename = '<template>'
            else:
                filename = encode_filename(filename)
            return self._compile(source, filename)
        except TemplateSyntaxError:
            exc_info = sys.exc_info()
        self.handle_exception(exc_info, source_hint=source_hint)

class Template(object):
    ...
    def render(self, *args, **kwargs):
        vars = dict(*args, **kwargs)
        try:
            return concat(self.root_render_func(self.new_context(vars)))
        except Exception:
            exc_info = sys.exc_info()
        return self.environment.handle_exception(exc_info, True)
    ...
```

这里也是通过compile对模板进行编译的

jinja2/parser.py

```
_statement_keywords = frozenset(['for', 'if', 'block', 'extends', 'print',
                                 'macro', 'include', 'from', 'import',
                                 'set', 'with', 'autoescape'])
_compare_operators = frozenset(['eq', 'ne', 'lt', 'lteq', 'gt', 'gteq'])

_math_nodes = {
   'add': nodes.Add,
   'sub': nodes.Sub,
   'mul': nodes.Mul,
   'div': nodes.Div,
   'floordiv': nodes.FloorDiv,
   'mod': nodes.Mod,
}
...
class Parser(object):
   def parse_statement(self):
       ...
       try:
           if token.value in _statement_keywords:
               return getattr(self, 'parse_' + self.stream.current.value)()
       ...
   def parse_print(self):
       node = nodes.Output(lineno=next(self.stream).lineno)
       node.nodes = []
       while self.stream.current.type != 'block_end':
           if node.nodes:
               self.stream.expect('comma')
           node.nodes.append(self.parse_expression())
       return node
```

这里面，print是个好东西，某些场景，限制了{{和}}的使用，只能使用{%和%}。这种清空，一般的想法是利用if来进行逻辑盲注，但是`{% print somedata %}`可以直接输出。

例题：网鼎杯 CTF 2018 第三场 Web mmmmy (暂无环境)

jinja2/defaults.py

```
DEFAULT_NAMESPACE = {
   'range':        range_type,
   'dict':         dict,
   'lipsum':       generate_lorem_ipsum,
   'cycler':       Cycler,
   'joiner':       Joiner,
   'namespace':    Namespace
}
```

默认的命名空间里面还有一些奇奇怪怪的对象存在的。

在探索测试的过程中发现，其实你随便输入一个字符串都是有用的。

比如`{{vvv}}`，一般情况你会发现页面空白啥的没有，但是加点东西就是新世界`{{vvv.__class__}}`。Jinja2对不存在的对象有一个特殊的定义Undefined类`

<class 'jinja2.runtime.Undefined'>`。

jinja2/runtime.py

```
@implements_to_string
class Undefined(object):
   ...
```

通过`{{ vvv.__class__.__init__.__globals__ }}`又能够搞事情了。

jinja2/filters.py

```
FILTERS = {
   'abs':                  abs,
   'attr':                 do_attr,
   'batch':                do_batch,
   'capitalize':           do_capitalize,
   'center':               do_center,
   'count':                len,
```

```
    'd':                    do_default,
    'default':              do_default,
    'dictsort':             do_dictsort,
    'e':                    escape,
    'escape':               escape,
    'filesizeformat':       do_filesizeformat,
    'first':                do_first,
    'float':                do_float,
    'forceescape':          do_forceescape,
    'format':               do_format,
    'groupby':              do_groupby,
    'indent':               do_indent,
    'int':                  do_int,
    'join':                 do_join,
    'last':                 do_last,
    'length':               len,
    'list':                 do_list,
    'lower':                do_lower,
    'map':                  do_map,
    'min':                  do_min,
    'max':                  do_max,
    'pprint':               do_pprint,
    'random':               do_random,
    'reject':               do_reject,
    'rejectattr':           do_rejectattr,
    'replace':              do_replace,
    'reverse':              do_reverse,
    'round':                do_round,
    'safe':                 do_mark_safe,
    'select':               do_select,
    'selectattr':           do_selectattr,
    'slice':                do_slice,
    'sort':                 do_sort,
    'string':               soft_unicode,
    'striptags':            do_striptags,
    'sum':                  do_sum,
    'title':                do_title,
    'trim':                 do_trim,
    'truncate':             do_truncate,
    'unique':               do_unique,
    'upper':                do_upper,
    'urlencode':            do_urlencode,
    'urlize':               do_urlize,
    'wordcount':            do_wordcount,
    'wordwrap':             do_wordwrap,
    'xmlattr':              do_xmlattr,
    'tojson':               do_tojson,
}
```

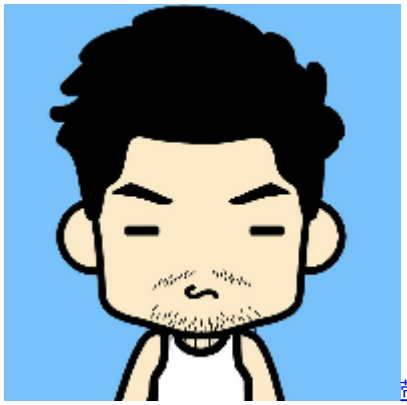
这些过滤器有些时候还是可以用到的，用法`{{ somedata | filter }}`,`{{url_for.__globals__.current_app.config|safe}}`

结束语

本文到这里就告一段落了，主要收获就是模板引擎命名空间或全局变量中的各种对象和函数。另外，其实还有很多地方没深入研究，大家有兴趣不妨翻翻源码找找有意思的事

标题有个(一)，算是给自己挖个坑，至于后续能不能填上就另说啦。。。。emmmmmmm

上一篇：Octopus恶意软件分析 下一篇：某cms两处sql注入分析
1. 2 条回复


带头小老弟 2018-10-17 16:48:22

讲的不是很清晰易懂。。。

0 回复Ta


virink 2018-10-17 17:41:09

@带头小老弟 QvQ。。。简单看看 render 的一些实现及上下文添加的函数和对象，感觉不难啊、、、看看代码就好了

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板