pocassist浅析

author: https://github.com/Ciyfly

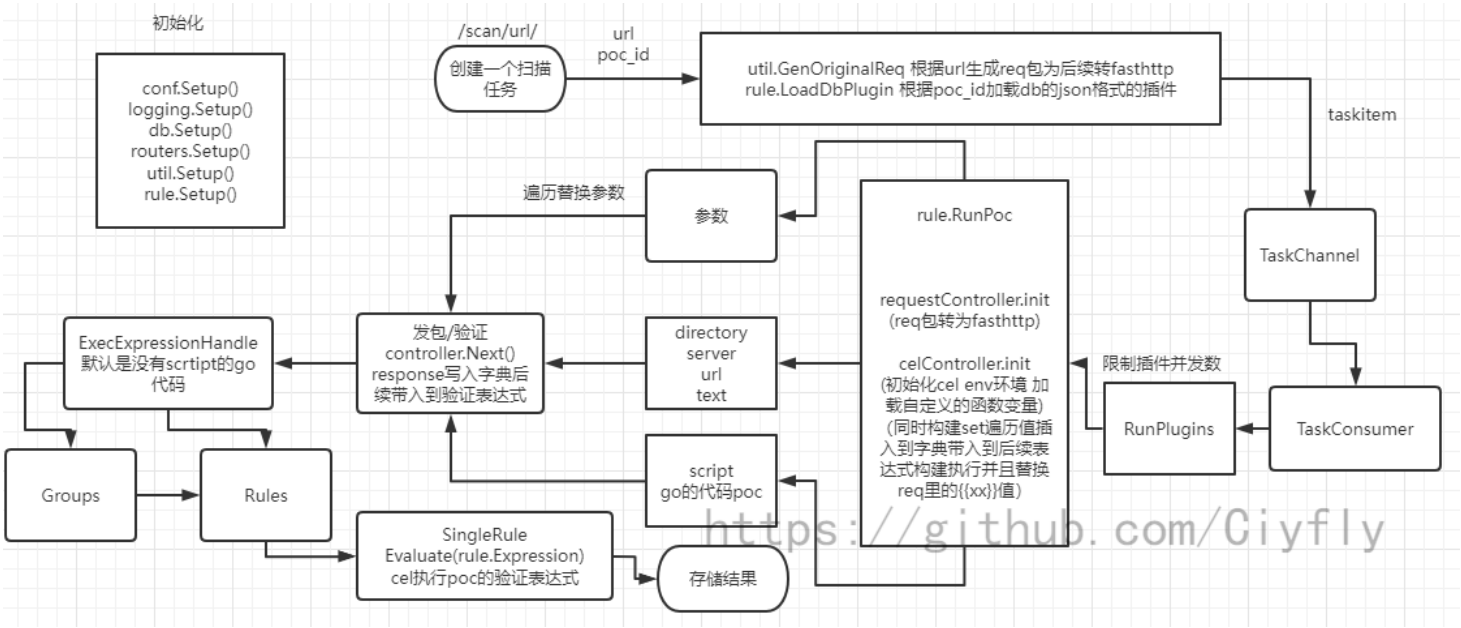
项目代码目录

```
.
├── api # web api的接口路由
│   ├── middleware
│   │   └── jwt
│   │       └── jwt.go
│   ├── msg
│   │   └── msg.go
│   └── routers
│       ├── route.go
│       ├── ui.go
│       └── v1
│           ├── auth
│           │   └── auth.go
│           ├── plugin
│           │   └── plugin.go
│           ├── scan
│           │   ├── result
│           │   │   └── result.go
│           │   ├── scan
│           │   │   └── scan.go
│           │   └── task
│           │       └── task.go
│           ├── vulnerability
│           │   └── vulnerability.go
│           └── webapp
│               └── webapp.go
├── cmd # 入口程序
│   └── pocassist.go
├── config.yaml # 全局配置
├── go.mod
├── go.sum
├── LICENSE
├── logs
│   └── pocassist.log
├── main.go
├── pkg # 核心代码 cel 解析 反连
│   ├── cel
│   │   ├── cel.go
│   │   ├── proto
│   │   │   ├── http.pb.go
│   │   │   └── http.proto
│   │   └── reverse
│   │       └── reverse.go
│   ├── conf
│   │   ├── config.go
│   │   └── default.go
│   ├── db
│   │   ├── auth.go
│   │   ├── conn.go
│   │   ├── plugin.go
│   │   ├── scanResult.go
│   │   ├── scanTask.go
│   │   ├── vulnerability.go
│   │   └── webapp.go
```

```
│   ├── file
│   │   └── file.go
│   ├── logging
│   │   └── log.go
│   └── util
│       ├── jwt.go
│       ├── rand.go
│       ├── request.go
│       ├── request_test.go
│       ├── result.go
│       ├── tcp.go
│       ├── util.go
│       └── version.go
├── poc
│   ├── rule # 处理poc的等
│   │   ├── cel.go
│   │   ├── content.go
│   │   ├── controller.go # 核心控制
│   │   ├── handle.go
│   │   ├── parallel.go
│   │   ├── request.go
│   │   ├── rule.go
│   │   └── run.go
│   └── scripts
│       ├── poc-go-dedecms-bakfile-disclosure.go
│       ├── poc-go-ecshop-anyone-login.go
│       ├── poc-go-elasticsearch-path-traversal.go
│       ├── poc-go-exim-cve-2017-16943-uaf.go
│       ├── poc-go-exim-cve-2019-15846-rce.go
│       ├── poc-go-fastcgi-file-read.go
│       ├── poc-go-ftp-unauth.go
│       ├── poc-go-iis-ms15034.go
│       ├── poc-go-jboss-console-weakpwd.go
│       ├── poc-go-jboss-invoker-servlet-rce.go
│       ├── poc-go-jboss-serialization.go
│       ├── poc-go-joomla-serialization.go
│       ├── poc-go-memcached-unauth.go
│       ├── poc-go-mongo-unauth.go
│       ├── poc-go-openssl-heartbleed.go
│       ├── poc-go-redis-unauth.go
│       ├── poc-go-rsync-anonymous.go
│       ├── poc-go-shiro-unserialize-550.go
│       ├── poc-go-smb-cve-2020-0796.go
│       ├── poc-go-tomcat-weak-pass.go
│       ├── poc-go-zookeeper-unauth.go
│       └── scripts.go
├── pocassist.db # poc db库
├── README.md
└── web # 前端资源
    ├── bindata.go
    └── build
        ├── asset-manifest.json
        ├── favicon.ico
        ├── index.html
        ├── manifest.json
```

```
├── precache-manifest.883d9a3cd99a61f6112882ff7a343fde.js
├── robots.txt
├── service-worker.js
└── static
    ├── css
    │   ├── 2.b2faedfb.chunk.css
    │   └── main.35003770.chunk.css
    ├── js
    │   ├── 2.b26af43a.chunk.js
    │   ├── 2.b26af43a.chunk.js.LICENSE.txt
    │   ├── main.c58b4be8.chunk.js
    │   └── runtime-main.89859971.js
    └── media
        └── bg.4bb50474.png
```

流程图



cmd/pocassist.go 整个程序启动入口

1. `init` 输出 banner等程序信息
2. `RunApp` 解析命令行指定的端口启动web程序

```
func init() {
	fmt.Printf("%s\n", conf.Banner)
	fmt.Printf("\t\tv" + conf.Version + "\n\n")
	fmt.Printf("\t\t" + conf.Website + "\n\n")
}

func InitAll() {
	// config 必须最先加载
	conf.Setup()
	logging.Setup()
	db.Setup()
	routers.Setup()
	util.Setup()
	rule.Setup()
}

// 使用viper 对配置热加载
func HotConf() {
	dir, err := filepath.Abs(filepath.Dir(os.Args[0]))
	if err != nil {
		log.Fatalf("cmd.HotConf, fail to get current path: %v", err)
	}
	// 配置文件路径 当前文件夹 + config.yaml
	configFile := path.Join(dir, conf.ConfigFileName)
	viper.SetConfigType("yaml")
	viper.SetConfigFile(configFile)
	// watch 监控配置文件变化
	viper.WatchConfig()
	viper.OnConfigChange(func(e fsnotify.Event) {
		// 配置文件发生变更之后会调用的回调函数
		log.Println("config file changed:", e.Name)
		InitAll()
	})
}

func RunApp() {
	app := cli.NewApp()
	app.Name = conf.ServiceName
	app.Usage = conf.Website
	app.Version = conf.Version

	app.Flags = []cli.Flag{
		&cli.StringFlag{
			// 后端端口
			Name:    "port",
			Aliases: []string{"p"},
			Value:   conf.DefaultPort,
			Usage:   "web server `PORT`",
		},
	}
	app.Action = RunServer
	err := app.Run(os.Args)
	if err != nil {
		log.Fatalf("cli.RunApp err: %v", err)
```

```
                return
        }
}

func RunServer(c *cli.Context) error {
        InitAll()
        HotConf()
        port := c.String("port")
        routers.InitRouter(port)
        return nil
}
```

init

go默认的 init 导入的是 `conf` 的一些值

conf `"github.com/jweny/pocassist/pkg/conf"`

一些banner version的信息 在 `pkg\conf\default.go`

```
func init() {
        fmt.Printf("%s\n", conf.Banner)
        fmt.Printf("\t\tv" + conf.Version + "\n\n")
        fmt.Printf("\t\t" + conf.Website + "\n\n")
}
```

RunApp

使用cli 解析命令行参数获取端口参数 调用 `RunServer` 启动web服务器

```
func RunApp() {
        app := cli.NewApp()
        app.Name = conf.ServiceName
        app.Usage = conf.Website
        app.Version = conf.Version

        app.Flags = []cli.Flag{
                &cli.StringFlag{
                        // 后端端口
                        Name:    "port",
                        Aliases: []string{"p"},
                        Value:   conf.DefaultPort,
                        Usage:   "web server `PORT`",
                },
        }
        app.Action = RunServer

        err := app.Run(os.Args)
        if err != nil {
                log.Fatalf("cli.RunApp err: %v", err)
                return
        }
}
```

RunServer

1. 初始化所有相关配置 `InitAll`
2. 对配置文件进行监听实现热加载 `HotConf`
3. 传递端口启动web服务 `routers.InitRouter(port)`

```
func RunServer(c *cli.Context) error {
        InitAll()
        HotConf()
        port := c.String("port")
        routers.InitRouter(port)
        return nil
}
```

InitAll 加载配置文件

```
func InitAll() {
        // config 必须最先加载
        conf.Setup()
        logging.Setup()
        db.Setup()
        routers.Setup()
        util.Setup()
        rule.Setup()
}
```

下面是所有的配置展开

conf.Setup() 加载全局配置文件

加载 `config.yaml` 配置文件

1. 拼接配置文件路径 不存在则初始化配置文件 `viper.ReadConfig(bytes.NewBuffer(defaultYamlByte))`
2. 读取配置文件 `viper.ReadInConfig()`

全局配置文件的结构体

```
type Config struct {
        HttpConfig    HttpConfig    `mapstructure:"httpConfig"`
        DbConfig      DbConfig      `mapstructure:"dbConfig"`
        PluginsConfig PluginsConfig `mapstructure:"pluginsConfig"`
        Reverse       Reverse       `mapstructure:"reverse"`
        ServerConfig  ServerConfig      `mapstructure:"serverConfig"`
        LogConfig         LogConfig             `mapstructure:"logConfig"`
}
```

```
func Setup() {
        dir, err := filepath.Abs(filepath.Dir(os.Args[0]))
        if err != nil {
                log.Fatalf("conf.Setup, fail to get current path: %v", err)
        }
        //配置文件路径 当前文件夹 + config.yaml
        configFile := path.Join(dir, "config.yaml")

        // 检测配置文件是否存在
        if !file.Exists(configFile) {
                WriteYamlConfig(configFile)
        }
        ReadYamlConfig(configFile)

}
```

logging.Setup() 加载日志配置文件

通过上面加载的全局配置文件
获取全局配置的日志级别 日志文件最大多少 备份多少 存多久等 使用的是 `go.uber.org/zap`

```
func Setup(){
        loggerCfg := conf.GlobalConfig.LogConfig
        NewLogger(conf.GlobalConfig.Level(), loggerCfg.MaxSize, loggerCfg.MaxBackups, loggerCfg.MaxAge,
                loggerCfg.Compress, conf.GlobalConfig)
}
```

db.Setup() 加载db配置文件

支持mysql和sqlite 赋值到 dbConfig里
并 通过 orm框架 `gorm` 创建 `GlobalDB` 并同步设置logger

```
func Setup() {
	var err error
	dbConfig := conf.GlobalConfig.DbConfig
	if conf.GlobalConfig.DbConfig.Sqlite == "" {
		// 配置mysql数据源
		if dbConfig.Mysql.User == "" ||
			dbConfig.Mysql.Password == "" ||
			dbConfig.Mysql.Host == "" ||
			dbConfig.Mysql.Port == "" ||
			dbConfig.Mysql.Database == "" {
			log.Fatalf("db.Setup err: config.yaml mysql config not set")
		}
		dsn := fmt.Sprintf("%s:%s@tcp(%s:%s)/%s?charset=utf8mb4&parseTime=True&loc=Local&timeout=%s",
			dbConfig.Mysql.User,
			dbConfig.Mysql.Password,
			dbConfig.Mysql.Host,
			dbConfig.Mysql.Port,
			dbConfig.Mysql.Database,
			dbConfig.Mysql.Timeout)

		GlobalDB, err = gorm.Open(mysql.Open(dsn), &gorm.Config{})
		if err != nil {
			log.Fatalf("db.Setup err: %v", err)
		}
	} else {
		// 配置sqlite数据源
		if dbConfig.Sqlite == "" {
			log.Fatalf("db.Setup err: config.yaml sqlite config not set")
		}
		if dbConfig.EnableDefault {
			dir, err := filepath.Abs(filepath.Dir(os.Args[0]))
			if err != nil {
				log.Fatalf("db.Setup, fail to get current path: %v", err)
			}
			//配置文件路径 当前文件夹 + config.yaml
			defaultSqliteFile := path.Join(dir, "pocassist.db")
			// 检测 sqlite 文件是否存在
			if !file.Exists(defaultSqliteFile) {
				log.Fatalf("db.Setup err: pocassist.db not exist, download at https://github.com/jweny/po
			}
		}

		GlobalDB, err = gorm.Open(sqlite.Open(dbConfig.Sqlite), &gorm.Config{
			DisableForeignKeyConstraintWhenMigrating: true,
		})
		if err != nil {
			log.Fatalf("db.Setup err: %v", err)
		}
	}

	if GlobalDB == nil {
		log.Fatalf("db.Setup err: db connect failed")
	}
```

```
        err = GlobalDB.AutoMigrate(&Auth{}, &Vulnerability{}, &Webapp{}, &Plugin{}, &Task{}, &Result{})

        if err != nil {
                log.Fatalf("db.Setup err: %v", err)
        }

        if conf.GlobalConfig.ServerConfig.RunMode == "release" {
                // release下
                GlobalDB.Logger = logger.Default.LogMode(logger.Silent)
        }
}
```

routers.Setup() 配置web

设置 gin的 mode级别 为 `release`

```
func Setup() {
        // gin 的【运行模式】运行时就已经确定 无法做到热加载
        gin.SetMode(conf.GlobalConfig.ServerConfig.RunMode)
}
```

gin 可设置的mode

```
const (
        // DebugMode indicates gin mode is debug.
        DebugMode = "debug"
        // ReleaseMode indicates gin mode is release.
        ReleaseMode = "release"
        // TestMode indicates gin mode is test.
        TestMode = "test"
)
```

util.Setup() 限流初始化 fasthttpclient jwt secret 初始化

```
func InitRate() {
        msQps := conf.GlobalConfig.HttpConfig.MaxQps  / 10
        limit := rate.Every(100 * time.Millisecond)
        limiter = rate.NewLimiter(limit, msQps)
}
```

使用的是 `golang.org/x/time/rate` 基于令牌桶算法
随着时间以 `1/r` 个令牌的速度向容积为b个令牌的桶中添加令牌 有请求就取走令牌 若令牌不足则不执行请求或者等待

通过该fasthttp创建client 是一个比 `net/http` 快10倍的 客户端请求库
去掉ua头
`DisablePathNormalizing` 是删除额外的斜杠，对特殊字符进行编码
并且配置代理

```
client := &fasthttp.Client{
    // If InsecureSkipVerify is true, TLS accepts any certificate
    TLSConfig: &tls.Config{InsecureSkipVerify: true},
    NoDefaultUserAgentHeader: true,
    DisablePathNormalizing:   true,
}
```

jwt secret

```
jwtSecret = []byte(conf2.GlobalConfig.ServerConfig.JwtSecret)
```

配置在 `config.yaml`

```
serverconfig:
  jwt_secret: pocassist
```

```
func Setup() {
        // 请求限速 limiter 初始化
        InitRate()
        // fasthttp client 初始化
        DownProxy := conf2.GlobalConfig.HttpConfig.Proxy
        client := &fasthttp.Client{
                // If InsecureSkipVerify is true, TLS accepts any certificate
                TLSConfig: &tls.Config{InsecureSkipVerify: true},
                NoDefaultUserAgentHeader: true,
                DisablePathNormalizing:   true,
        }
        if DownProxy != "" {
                log.Info("[fasthttp client use proxy ]", DownProxy)
                client.Dial = fasthttpproxy.FasthttpHTTPDialer(DownProxy)
        }

        fasthttpClient = client

        // jwt secret 初始化
        jwtSecret = []byte(conf2.GlobalConfig.ServerConfig.JwtSecret)
}
```

rule.Setup() 规则配置相关初始化

`poc\rule\handle.go`

1. 初始化一个 字典 key是字符串 value是 HandlerFunc列表
2. 给 `Handles` 添加 `"script"` 的value是 函数 `ExecScriptHandle` ExecScriptHandle 函数见下面详细展开
3. 给 `Handles` 添加 `"appendparam"` 的value是 函数 `ExecExpressionHandle` ExecExpressionHandle 函数见下面详细展开
4. 给 `Handles` 添加 `"replaceparam"` 的value是 函数 `ExecExpressionHandle`
5. 调用函数 `InitTaskChannel` 函数见下面详细展开

```
func Setup() {
        Handles = make(map[string][]HandlerFunc)
        Handles[AffectScript] = []HandlerFunc{ExecScriptHandle}
        Handles[AffectAppendParameter] = []HandlerFunc{ExecExpressionHandle}
        Handles[AffectReplaceParameter] = []HandlerFunc{ExecExpressionHandle}
        InitTaskChannel()
}
```

HandlerFunc 定义了一个统一的handler函数规范 参数是 `controllerContext`

poc/rule/handle.go#9

```
type HandlerFunc func(ctx controllerContext)
```

poc/rule/handle.go#52

controllerContext 定义了一个接口 后面应该有多个实现控制器吧
然后 会有多个 HandlerFunc 函数解析多种 controllerContext接口实现的结构体

```
type controllerContext interface {
        Next()
        Abort()
        IsAborted() bool
        GetString(key string) string
        Set(key string, value interface{})
        Get(key string) (value interface{}, exists bool)
        GetPoc() *Poc
        Groups(bool) (result bool, err error)
        Rules([]Rule, bool) (result bool, err error)
        GetPocName() string
        GetOriginalReq() *http.Request
        SetResult(result *util.ScanResult)
        IsDebug() bool
        // RegisterHandle(f HandlersChain)
}
```

ExecScriptHandle 是poc/script下的poc Handler函数

1. 通过控制器上下文 其实就是一个poc结构体 获取名称
2. 使用poc名称 通过 `scripts.GetScriptFunc` 获取扫描函数
   GetScriptFunc的函数是从 `scriptHandlers` 字典中 通过 `pocNmae` 获取 `scanFunc` 即是验证方法
3. 输出信息日志
4. 处理端口 和是否是https 创建 `scripts.ScriptScanArgs` 脚本扫描函数使用的参数结构体 传入 `scanFunc`
5. 执行后将结果传递给 `SetResult` 保存结果 并执行 调用 `Abort` 终止

```
func ExecScriptHandle(ctx controllerContext) {
	pocName := ctx.GetPocName()
	scanFunc := scripts.GetScriptFunc(pocName)
	if scanFunc == nil {
		log.Error("[rule/handle.go:ExecScriptHandle error] ", "scan func is nil")
		ctx.Abort()
		return
	}
	log.Info("[rule/handle.go:ExecScriptHandle script start]" + pocName)

	var isHTTPS bool
	// 处理端口
	defaultPort := 80
	originalReq := ctx.GetOriginalReq()
	if originalReq == nil {
		log.Error("[rule/handle.go:ExecScriptHandle error] ", "original request is nil")
		ctx.Abort()
		return
	}

	if originalReq.URL.Scheme == "https" {
		isHTTPS = true
		defaultPort = 443
	}

	if originalReq.URL.Port() != "" {
		port, err := strconv.ParseUint(originalReq.URL.Port(), 10, 16)
		if err != nil {
			ctx.Abort()
			return
		}
		defaultPort = int(port)
	}

	args := &scripts.ScriptScanArgs{
		Host:    originalReq.URL.Hostname(),
		Port:    uint16(defaultPort),
		IsHTTPS: isHTTPS,
	}
	result, err := scanFunc(args)
	if err != nil {
		log.Error("[rule/handle.go:ExecScriptHandle error] ", err)
		ctx.Abort()
		return
	}
	ctx.SetResult(result)
	ctx.Abort()
}
```

ExecExpressionHandle 是db中的yaml的类似xray格式的poc Handler函数

1. 获取Poc

2. poc的 `Groups` 如果不是空 `ctx.Groups(ctx.IsDebug())`
3. 如果是空 `ctx.Rules(poc.Rules,ctx.IsDebug())`
4. 返回的result结果不是空 调用 `Abort` 终止否则返回

```
func ExecExpressionHandle(ctx controllerContext){
	var result bool
	var err error

	poc := ctx.GetPoc()
	if poc == nil {
		log.Error("[rule/handle.go:ExecExpressionHandle error] ", "poc is nil")
		return
	}
	if poc.Groups != nil {
		result, err = ctx.Groups(ctx.IsDebug())
	} else {
		result, err = ctx.Rules(poc.Rules,ctx.IsDebug())
	}
	if err != nil {
		log.Error("[rule/handle.go:ExecExpressionHandle error] ", err)
		return
	}
	if result {
		ctx.Abort()
	}
	return
}
```

InitTaskChannel 任务管道

初始化任务管道
限制管道大小10个
管道里的数据是 `TaskItem`

```
// 限制并发
type TaskItem struct {
	OriginalReq *http.Request // 原始请求
	Plugins     []Plugin      // 检测插件
	Task        *db.Task      // 所属任务
}

var TaskChannel chan *TaskItem

func InitTaskChannel(){
	// channel 限制 target 并发
	concurrent := 10
	if conf.GlobalConfig.PluginsConfig.Concurrent != 0 {
		concurrent = conf.GlobalConfig.PluginsConfig.Concurrent
	}
	TaskChannel = make(chan *TaskItem, concurrent)
}
```

HotConf 配置热加载

配置配置文件路径 当 配置文件发生任何事件 都重新调用 `InitAll` 重新初始化配置

```
// 使用viper 对配置热加载
func HotConf() {
        dir, err := filepath.Abs(filepath.Dir(os.Args[0]))
        if err != nil {
                log.Fatalf("cmd.HotConf, fail to get current path: %v", err)
        }
        // 配置文件路径 当前文件夹 + config.yaml
        configFile := path.Join(dir, conf.ConfigFileName)
        viper.SetConfigType("yaml")
        viper.SetConfigFile(configFile)
        // watch 监控配置文件变化
        viper.WatchConfig()
        viper.OnConfigChange(func(e fsnotify.Event) {
                // 配置文件发生变更之后会调用的回调函数
                log.Println("config file changed:", e.Name)
                InitAll()
        })
}
```

routers.InitRouter 初始化web的路由并启动

通过 `gin` 来启动web服务

1. 如果是 `debug` 模式 开启 `swagger`
2. 设置静态资源路径 `ui`
3. 定义api 增删改查接口

```
func InitRouter(port string) {
	router := gin.Default()

	// debug 模式下 开启 swagger
	if conf.GlobalConfig.ServerConfig.RunMode == "debug" {
		router.GET("/swagger/*any", gs.WrapHandler(swaggerFiles.Handler))
	}

	// ui
	router.StaticFS("/ui", BinaryFileSystem("web/build"))
	router.GET("/", func(c *gin.Context) {
		c.Redirect(http.StatusPermanentRedirect, "/ui")
	})

	// api
	v1 := router.Group("/api/v1")
	{
		v1.POST("/user/login", auth.Login)
		userRoutes := v1.Group("/user")
		userRoutes.Use(jwt.JWT())
		{
			userRoutes.POST("/self/resetpwd/", auth.Reset)
			userRoutes.GET("/info", auth.Self)
			userRoutes.GET("/logout", auth.Logout)
		}

		pluginRoutes := v1.Group("/poc")
		pluginRoutes.Use(jwt.JWT())
		{
			// all
			pluginRoutes.GET("/", plugin.Get)
			// 增
			pluginRoutes.POST("/", plugin.Add)
			// 改
			pluginRoutes.PUT("/:id/", plugin.Update)
			// 详情
			pluginRoutes.GET("/:id/", plugin.Detail)
			// 删
			pluginRoutes.DELETE("/:id/", plugin.Delete)
			// 测试单个poc
			pluginRoutes.POST("/run/", plugin.Run)
			// 上传yaml文件
			pluginRoutes.POST("/upload/", plugin.UploadYaml)
			// 下载yaml文件
			pluginRoutes.POST("/download/", plugin.DownloadYaml)
		}

		vulRoutes := v1.Group("/vul")
		vulRoutes.Use(jwt.JWT())
		{
			// basic
			vulRoutes.GET("/basic/", vulnerability.Basic)
			// all
			vulRoutes.GET("/", vulnerability.Get)
```

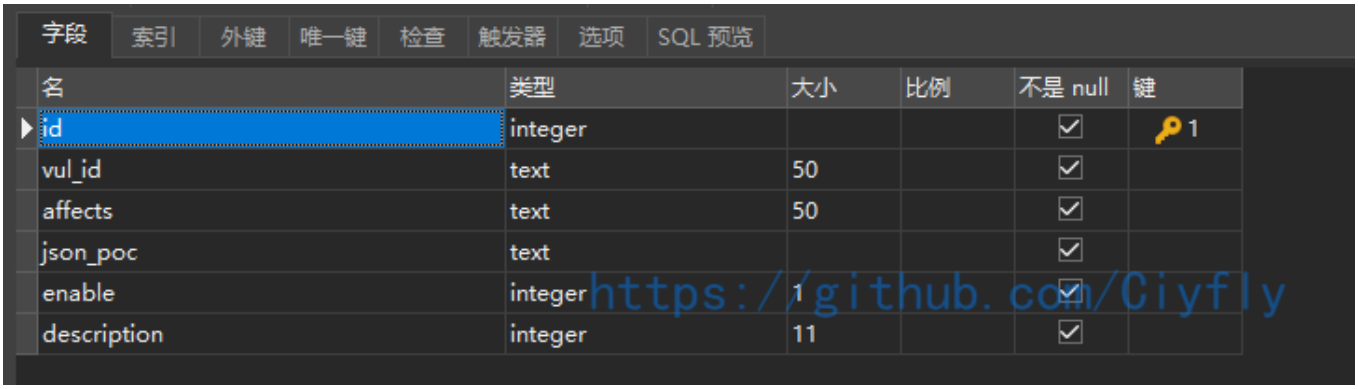
```
                // 增
                vulRoutes.POST("/", vulnerability.Add)
                // 改
                vulRoutes.PUT("/:id/", vulnerability.Update)
                // 详情
                vulRoutes.GET("/:id/", vulnerability.Detail)
                // 删
                vulRoutes.DELETE("/:id/", vulnerability.Delete)
        }

        appRoutes := v1.Group("/product")
        appRoutes.Use(jwt.JWT())
        {
                // all
                appRoutes.GET("/", webapp.Get)
                // 增
                appRoutes.POST("/", webapp.Add)
                // 改
                appRoutes.PUT("/:id/", webapp.Update)
                // 详情
                appRoutes.GET("/:id/", webapp.Detail)
                // 删
                appRoutes.DELETE("/:id/", webapp.Delete)
        }

        scanRoutes := v1.Group("/scan")
        scanRoutes.Use(jwt.JWT())
        {
                scanRoutes.POST("/url/", scan2.Url)
                scanRoutes.POST("/raw/", scan2.Raw)
                scanRoutes.POST("/list/", scan2.List)
        }

        taskRoutes := v1.Group("/task")
        taskRoutes.Use(jwt.JWT())
        {
                // all
                taskRoutes.GET("/", task.Get)
                // 删
                taskRoutes.DELETE("/:id/", task.Delete)
        }

        resultRoutes := v1.Group("/result")
        resultRoutes.Use(jwt.JWT())
        {
                // all
                resultRoutes.GET("/", result.Get)
                // 删
                resultRoutes.DELETE("/:id/", result.Delete)
        }

}
router.Run(":" + port)
log.Info("server start at port:", port)
```

```
}
```

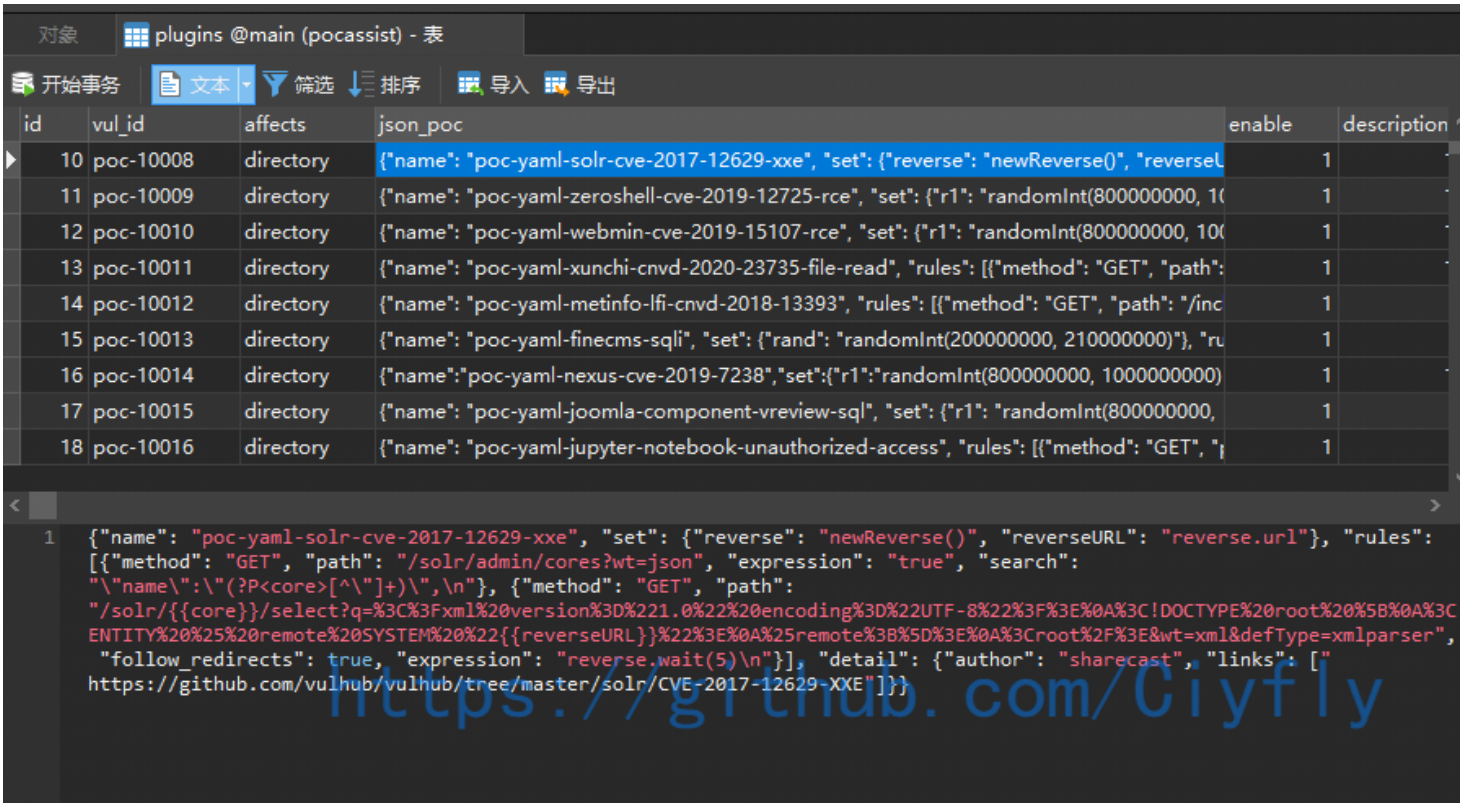
这样web都已经启动了 那我们看看如果再界面上创建一个扫描任务是怎么样的

db文件

如果程序要想启动还需要个db文件 地址 `https://github.com/jweny/pocassistdb`
里面包含web需要使用的账户 数据 描述等 最主要是的表 `plugins` 如下图所示



他是把yaml格式的poc解析成json后加上描述名称等字段入库



创建一个扫描任务

首先启动程序后 配置文件默认监听是 `1321` 端口

看后端代码

`api\routers\route.go#107`

```
scanRoutes := v1.Group("/scan")
{
    // url的是创建单个url的任务
    scanRoutes.POST("/url/", scan2.Url)
    // raw是 上传raw文件的任务
    scanRoutes.POST("/raw/", scan2.Raw)
    // list是url列表文件的任务
    scanRoutes.POST("/list/", scan2.List)
}
```

url的处理函数

1. 接收前端参数获取要用的poc名称 目标等信息
2. 通过 `util.GenOriginalReq` 生成一个请求包
3. 通过 `rule.LoadDbPlugin` 从数据库中加载poc 全部的poc还是指定的poc 根据vul_id来查询
4. 创建任务 `task` 创建 `taskItem` 包含生成的请求包 poc列表 task
5. `rule.TaskProducer(taskItem)`

```
func Url(c *gin.Context) {
	scan := scanSerializer{}
	err := c.ShouldBindJSON(&scan)
	if err != nil {
		c.JSON(msg.ErrResp("测试url不可为空，扫描类型为multi或all"))
		return
	}

	oreq, err := util.GenOriginalReq(scan.Target)
	if err != nil || oreq == nil {
		c.JSON(msg.ErrResp("原始请求生成失败"))
		return
	}

	// 插件列表
	plugins, err := rule.LoadDbPlugin(scan.Type, scan.VulList)
	if err != nil || plugins == nil{
		c.JSON(msg.ErrResp("poc插件加载失败" + err.Error()))
		return
	}
	token := c.Request.Header.Get("Authorization")
	claims, _ := util.ParseToken(token)

	// 创建任务
	task := db.Task{
		Operator: claims.Username,
		Remarks: scan.Remarks,
		Target:  scan.Target,
	}
	db.AddTask(&task)
	taskItem := &rule.TaskItem{
		OriginalReq: oreq,
		Plugins:     plugins,
		Task:        &task,
	}

	c.JSON(msg.SuccessResp("任务下发成功"))
	go rule.TaskProducer(taskItem)
	go rule.TaskConsumer()
	return
}
```

GenOriginalReq 生成一个原始请求

`pkg\util\request.go#421`

这个request对象是没有发起请求的

1. 校验目标是否可以连接 通过发一个tcp包来判断 排除 `icmp` 的
2. 生成一个request对象 `originalReq, err := http.NewRequest("GET", fixTarget, nil)`
3. 添加 `headers` `host` `Accept-Encoding` `Connection` 等 并返回生成的request包

```
func GenOriginalReq(target string) (*http.Request, error) {
        verify, fixTarget := VerifyInputTarget(target)
        if !verify {
                errMsg := fmt.Errorf("util/requests.go:GenOriginalReq %s can not connect", target)
                log.Error(errMsg)
                return nil, errMsg
        }
        originalReq, err := http.NewRequest("GET", fixTarget, nil)
        if err != nil {
                errMsg := fmt.Errorf("util/requests.go:GenOriginalReq %s original request gen error %v", target, err)
                log.Error(errMsg)
                return nil, errMsg
        }
        originalReq.Header.Set("Host", originalReq.Host)
        originalReq.Header.Set("Accept-Encoding", "gzip, deflate")
        originalReq.Header.Set("Accept", "*/*")
        originalReq.Header.Set("User-Agent", conf.GlobalConfig.HttpConfig.Headers.UserAgent)
        originalReq.Header.Set("Accept-Language", "en")
        originalReq.Header.Set("Connection", "close")
        return originalReq, nil
}
```

rule.TaskProducer 将taskitem写到TaskChannel管道里

```
func TaskProducer(item *TaskItem) {
        TaskChannel <- item
}
```

rule.TaskConsumer() 消费TaskChannel管道里的数据调用poc测试

从 `TaskChannel` 管道中接收数据后 通过 `RunPlugins` 处理 数据

```
func TaskConsumer() {
        for item := range TaskChannel {
                // 校验格式
                err := item.Verify()
                if err != nil {
                        log.Error("[rule/poc.go:WriteTaskError scan error] ", err)
                        db.ErrorTask(item.Task.Id)
                        continue
                }
                RunPlugins(item)
        }
}
```

RunPlugins 协程限制并发运行插件

poc\rule\parallel.go#137

1. 从配置文件中获取到限制并发数量 `8`
2. 通过 `ants` 创建协程池来管理协程
3. 通过 `rule.RunPoc` 来调用poc
4. 当所有poc都run完 才认为这个任务是完成的了

```
func RunPlugins(item *TaskItem) {
        // 限制插件并发数
        var wg sync.WaitGroup
        parallel := conf.GlobalConfig.PluginsConfig.Parallel
        p, _ := ants.NewPoolWithFunc(parallel, func(item interface{}) {
                RunPoc(item, false)
                wg.Done()
        })
        defer p.Release()
        oreq := item.OriginalReq
        plugins := item.Plugins
        task := item.Task

        log.Info("[rule/parallel.go:TaskConsumer start scan]", oreq.URL.String())
        for i := range plugins {
                item := &ScanItem{oreq, &plugins[i], task}
                wg.Add(1)
                p.Invoke(item)
        }
        wg.Wait()
        db.DownTask(task.Id)
}
```

rule.RunPoc 运行poc

`poc\rule\run.go#54`

根据poc的参数位置替换请求对应位置的参数 如果script即go的poc代码直接调用 如果json的即原来是yaml的在db里的

通过cel控制器处理 验证 最后输出结果

1. 获取 `scanItem` 后调用 `Verify` 方法 只是验证数据是否都不是空的
2. 定义 req控制器 `RequestController` 定义 cel控制器 `CelController`
3. 初始化 req `requestController.Init(scanItem.OriginalReq)` 生成Fasthttp 原始请求转为fasthttp
4. 获取handler `handles := getHandles(scanItem.Plugin.Affects)` 默认是数据库里的 除 `script` 的即go脚本的
5. 初始化cel `celController.Init(scanItem.Plugin.JsonPoc)` 对这个poc进行cel构建
6. 根据实际的req包与 poc合并处理
   `celController.InitSet(scanItem.Plugin.JsonPoc, requestController.New)` 将poc的set变量赋值 赋值request 到 cel控制器中传递变量后给后续表达式 使用
7. `util.RequestPut(requestController.New)` 将fasthttp请求包写到 sync.pool 共享池里节省内存

8. 根据数据库中的插件的影响类型 例如是参数类型 目录 脚本等类型分别处理 这个类型是数据库里的表 `plugins` 的字段 `affects`
   根据给出的db文件 类型只有 `directory` 即 `AffectDirectory` 和script的
9. 如果是参数类型
   a. 通过 `InitOriginalQueryParams` 获取参数数据 如果get就获取params否则就获取body
   b. 初始化poc控制器 通过
   ```
   InitPocController(&requestController, scanItem.Plugin, &celController, handles)
   ```
   c. 获取所有url参数
   ```
   originalParamFields, err := url.ParseQuery(requestController.OriginalQueryParams)
   ```
   c. 遍历参数 `requestController.FixQueryParams(field, payload, controller.Plugin.Affects)` 将payload插入到所有的参数中
   d. `controller.Next()` 对poc中的每个HandleFunc函数进行调用
   e. 如果被终止了说明存在漏洞 封装漏洞结果 `util.VulnerableHttpResult` 写任务结果 `WriteTaskResult`
   f. 重置这个控制器 写到 poc控制器池 `ControllerPool` 里 节省内存给下一次调用
   g. 如果没漏洞也要写到 poc控制器池 `ControllerPool` 里
10. 如果是 `directory` `server` `url` `text` 级别的
    a. 初始化控制器 `InitPocController(&requestController, scanItem.Plugin, &celController, handles)`
    b. 调用 `Next` 方法
    c. 判断是否存在漏洞同上
    d. 如果是debug状态没漏洞的话 结果是
    ```
    util.DebugVulnerableHttpResult(controller.GetOriginalReq().URL.String(), "", controller.Request.Raw)
    ```
    封装的
11. 如果是 `script`
    a. 初始化poc控制器
    ```
    controller := InitPocController(&requestController, scanItem.Plugin, &celController, handles)
    ```
    b. 如果是终止的 脚本结果不是空 脚本的漏洞不是空 则保存结果 `util.ScanResult`
    c. 写漏洞结构 poc控制器重置入池
    d. 如果是debug模式 也是使用 `util.DebugVulnerableHttpResult` 封装下结果
    e. 如果没漏洞也要写到 poc控制器池 `ControllerPool` 里
12. 都不是的话 返回没有漏洞 `util.InVulnerableResult`

```
// 执行单个poc
func RunPoc(inter interface{}, debug bool) (result *util.ScanResult, err error) {
	scanItem := inter.(*ScanItem)
	err = scanItem.Verify()
	if err != nil {
		log.Error("[rule/poc.go:RunPoc scan item verify error] ", err)
		return nil, err
	}
	log.Info("[rule/poc.go:RunPoc current plugin]", scanItem.Plugin.JsonPoc.Name)

	var requestController RequestController
	var celController CelController

	err = requestController.Init(scanItem.OriginalReq)
	if err != nil {
		log.Error("[rule/poc.go:RunPoc request controller init error] ", err)
		return nil, err
	}

	handles := getHandles(scanItem.Plugin.Affects)

	err = celController.Init(scanItem.Plugin.JsonPoc)
	if err != nil {
		log.Error("[rule/poc.go:RunPoc cel controller init error] ", err)
		return nil, err
	}

	err = celController.InitSet(scanItem.Plugin.JsonPoc, requestController.New)
	if err != nil {
		util.RequestPut(requestController.New)
		log.Error("[rule/poc.go:RunPoc cel controller init set error] ", err)
		return nil, err
	}
	switch scanItem.Plugin.Affects {
	// 影响为参数类型
	case AffectAppendParameter, AffectReplaceParameter:
		{
			err := requestController.InitOriginalQueryParams()
			if err != nil {
				log.Error("[rule/poc.go:RunPoc init original request params error] ", err)
				return nil, err
			}
			controller := InitPocController(&requestController, scanItem.Plugin, &celController, handles)
			controller.Debug = debug
			paramPayloadList := scanItem.Plugin.JsonPoc.Params

			originalParamFields, err := url.ParseQuery(requestController.OriginalQueryParams)
			if err != nil {
				log.Error("[rule/poc.go:RunPoc params query parse error] ", err)
				return nil, err
			}

			for field := range originalParamFields {
				for _, payload := range paramPayloadList {
```

```
                                log.Info("[rule/poc.go:RunPoc param payload]", payload)
                                err = requestController.FixQueryParams(field, payload, controller.Plugin.Affects)
                                if err != nil {
                                        log.Error("[rule/poc.go:RunPoc fix request params error] ", err)
                                        continue
                                }
                                controller.Next()
                                if controller.IsAborted() {
                                        // 存在漏洞
                                        result = util.VulnerableHttpResult(controller.GetOriginalReq().URL.String
                                        WriteTaskResult(scanItem, result)
                                        PutController(controller)
                                        return result, nil
                                }
                                controller.Index = 0
                        }
                }
                // 没漏洞
                result = &util.InVulnerableResult
                PutController(controller)
                return result, nil
        }
case AffectDirectory, AffectServer, AffectURL, AffectContent:
        {
                controller := InitPocController(&requestController, scanItem.Plugin, &celController, handles)
                controller.Debug = debug
                controller.Next()
                if controller.IsAborted() {
                        // 存在漏洞
                        result = util.VulnerableHttpResult(controller.GetOriginalReq().URL.String(), "", controll
                        WriteTaskResult(scanItem, result)
                        PutController(controller)
                        return result, nil
                } else if debug{
                        // debug 没漏洞
                        result = util.DebugVulnerableHttpResult(controller.GetOriginalReq().URL.String(), "", con
                        PutController(controller)
                        return result, nil
                }else {
                        // 没漏洞
                        result = &util.InVulnerableResult
                        PutController(controller)
                        return result, nil
                }
        }
case AffectScript:
        {
                controller := InitPocController(&requestController, scanItem.Plugin, &celController, handles)
                controller.Debug = debug
                controller.Next()
                if controller.IsAborted() && controller.ScriptResult != nil && controller.ScriptResult.Vulnerable
                        // 存在漏洞 保存结果
                        result = &util.ScanResult{
                                Vulnerable: controller.ScriptResult.Vulnerable,
                                Target:     controller.ScriptResult.Target,
```

```
                                        Output:     controller.ScriptResult.Output,
                                        ReqMsg:     controller.ScriptResult.ReqMsg,
                                        RespMsg:    controller.ScriptResult.RespMsg,
                                }
                                WriteTaskResult(scanItem, controller.ScriptResult)
                                PutController(controller)
                                return result, nil
                        } else if debug {
                                // debug 没漏洞
                                result = util.DebugVulnerableHttpResult(controller.GetOriginalReq().URL.String(), "", con
                                PutController(controller)
                                return result, nil
                        } else {
                                // 没漏洞
                                PutController(controller)
                                return &util.InVulnerableResult, nil
                        }
                }
        }
        // 默认返回没有漏洞
        return &util.InVulnerableResult, nil
}
```

celController.Init cel控制器初始化

poc\rule\cel.go#25

1. `cel2.InitCelOptions()` 加入 大量的xray的自定义函数 例如 `randomInt` `base64Decode` 等 还有变量
2. `option.AddRuleSetOptions(poc.Set)` 注入set的变量 set的是 poc里的定义的变量 类型默认都是字符串的
3. `cel2.InitCelEnv(&option)` 创建env环境
4. 将cel的env赋值到 `Env` 并定义个参数map `ParamMap` 到cel控制器中

```
//      初始化
func (cc *CelController) Init(poc *Poc) (err error) {
        //      1.生成cel env环境
        option := cel2.InitCelOptions()
        //      注入set定义的变量
        if poc.Set != nil {
                option.AddRuleSetOptions(poc.Set)
        }
        env, err := cel2.InitCelEnv(&option)
        if err != nil {
                log.Error("[rule/cel.go:Init init cel env error]", err)
                return err
        }
        cc.Env = env
        // 初始化变量列表
        cc.ParamMap = make(map[string]interface{})
        return nil
}
```

celController.InitSet cel控制器初始化

poc\rule\cel.go#44

1. 将request添加到 ParamMap
2. 将set变量也添加到 ParamMap 如果是反连要创建个反连 reverse.NewReverse()
3. cel2.Evaluate(cc.Env, value, cc.ParamMap) 构建set的cel 执行
4. cel执行解析后将对于解析后的值替换回 ParamMap 为了给后续poc表达式使用变量

```
// 处理poc: set
func (cc *CelController) InitSet(poc *Poc, newReq *proto.Request) (err error) {
        // 如果没有set 就直接返回
        if len(poc.Set) == 0 {
                return
        }
        cc.ParamMap["request"] = newReq

        for _, setItem := range poc.Set {
                key := setItem.Key.(string)
                value := setItem.Value.(string)
                // 反连平台
                if value == "newReverse()" {
                        cc.ParamMap[key] = reverse.NewReverse()
                        continue
                }
                out, err := cel2.Evaluate(cc.Env, value, cc.ParamMap)
                if err != nil {
                        return err
                }
                switch value := out.Value().(type) {
                // set value 无论是什么类型都先转成string
                case *proto.UrlType:
                        cc.ParamMap[key] = util.UrlTypeToString(value)
                case int64:
                        cc.ParamMap[key] = int(value)
                default:
                        cc.ParamMap[key] = fmt.Sprintf("%v", out)
                }
        }
        return
}
```

controller.Next()

poc\rule\controller.go#221

这里的 controller.Handles 是 handles := getHandles(scanItem.Plugin.Affects) 根据类型获取到的poc Handler 即是 所有除go代码的poc

计算一共需要测试多个Handles 遍历调用

```
func (controller *PocController) Next() {
        for controller.Index < int64(len(controller.Handles)) {
                controller.Handles[controller.Index](controller)
                controller.Index++
        }
}
```

因为通过 `handles := getHandles(scanItem.Plugin.Affects)` 这里默认会添加的是 `ExecExpressionHandle`

ExecExpressionHandle

`poc\rule\handle.go#15`

1. `ctx.GetPoc()` 获取当前测试的poc 即 `controller.Plugin.JsonPoc`
2. 如果poc有 `Groups` 那么调用 `ctx.Groups(ctx.IsDebug())` 否则调用 `ctx.Rules(poc.Rules,ctx.IsDebug())`
   有 Groups的poc举例

```yaml
params: []
name: poc-yaml-shopxo-cnvd-2021-15822
set: {}
rules: []
groups:
  Linux:
  - method: GET
    path: /public/index.php?s=/index/qrcode/download/url/L2V0Yy9wYXNzd2Q=
    headers: {}
    body: ""
    search: ""
    followredirects: false
    expression: |
      response.status == 200 && "root:[x*]:0:0:".bmatches(response.body)
  Windows:
  - method: GET
    path: /public/index.php?s=/index/qrcode/download/url/L1dpbmRvd3Mvd2luLmluaQ=
    headers: {}
    body: ""
    search: ""
    followredirects: false
    expression: |
      response.status == 200 && response.body.bcontains(b"extensions") && response.body.bcontains(b"for
detail:
  author: ""
  links: []
  description: ""
  version: ""
```

不是 Groups的是 Rules的举例

```
{
        "name": "poc-yaml-solr-cve-2017-12629-xxe",
        "set": {
                "reverse": "newReverse()",
                "reverseURL": "reverse.url"
        },
        "rules": [{
                "method": "GET",
                "path": "/solr/admin/cores?wt=json",
                "expression": "true",
                "search": "\"name\":\"(?P<core>[^\"]+)\",\n"
        }, {
                "method": "GET",
                "path": "/solr/{{core}}/select?q=%3C%3Fxml%20version%3D%221.0%22%20encoding%3D%22UTF-8%
                "follow_redirects": true,
                "expression": "reverse.wait(5)\n"
        }],
        "detail": {
                "author": "sharecast",
                "links": ["https://github.com/vulhub/vulhub/tree/master/solr/CVE-2017-12629-XXE"]
        }
}
```

```
func ExecExpressionHandle(ctx controllerContext){
        var result bool
        var err error
        poc := ctx.GetPoc()
        if poc == nil {
                log.Error("[rule/handle.go:ExecExpressionHandle error] ", "poc is nil")
                return
        }
        if poc.Groups != nil {
                result, err = ctx.Groups(ctx.IsDebug())
        } else {
                result, err = ctx.Rules(poc.Rules,ctx.IsDebug())
        }
        if err != nil {
                log.Error("[rule/handle.go:ExecExpressionHandle error] ", err)
                return
        }
        if result {
                ctx.Abort()
        }
        return
}
```

controller.Groups 含有 Groups的poc执行

`poc\rule\controller.go#205`

1. 遍历 groups里的rules

2. poc控制器调用 `controller.Rules(rules, debug)` 返回规则结果 如果一个rules成功即 返回成功 这个成功是表示rule的表达式也是对的 如果整个rules的表达式都是true就是对的
   下面我们详细看下 `controller.Rules`
3. `group` 只要有一个rules成功即返回成功否则返回false

```
// 执行 groups
func (controller *PocController) Groups(debug bool) (bool, error) {
        groups := controller.Plugin.JsonPoc.Groups
        // groups 就是多个rules 任何一个rules成功 即返回成功
        for _, rules := range groups {
                rulesResult, err := controller.Rules(rules, debug)
                if err != nil || !rulesResult {
                        continue
                }
                // groups中一个rules成功 即返回成功
                if rulesResult {
                        return rulesResult, nil
                }
        }
        return false, nil
}
```

controller.Rules 含有 Rules的poc执行

`poc\rule\controller.go#186`

如果poc含有多个rule的情况 遍历调用 `controller.SingleRule`

```
// 执行 rules
func (controller *PocController) Rules(rules []Rule, debug bool) (bool, error) {
        success := false
        for _, rule := range rules {
                singleRuleResult, err := controller.SingleRule(&rule, debug)
                if err != nil {
                        log.Error("[rule/controller.go:Rules run error]", err)
                        return false, err
                }
                if !singleRuleResult {
                        //如果false 直接跳出循环 返回
                        success = false
                        break
                }
                success = true
        }
        return success, nil
}
```

controller.SingleRule 单条rule怎么执行

`poc\rule\controller.go#148`

1. 调用rule的 `Verify` 限制rule中的path必须以 `"/"` 开头
2. `rule.ReplaceSet(controller.CEL.ParamMap)` 替换set对应的值 将请求中 `headers` `path` `body` 中 `{{xxx}}` 的xxx根据之前cel解析构建的变量替换
3. 根据原始请求 + rule 生成并发起新的请求 http `controller.DoSingleRuleRequest` 返回resp
4. 给 `controller.CEL.ParamMap["response"]` 赋值为 返回的resp
5. 如果rule Search不是空的进行匹配 `rule.Search` 是正则字符串 匹配body 并返回结果map
6. `Evaluate` 调用rule的表达式 返回表达式的结果是布尔值 `cel2.Evaluate(cc.Env, char, cc.ParamMap)` 计算表达式的时候是将 最开始的 `request` 以及刚刚通过 `controller.DoSingleRuleRequest` 返回后的 `response` 以及set set是之前执行表达式解析获取到 这个 `ParamMap` 都带入进去构建解析poc的验证表达式了
7. 如果debug或者rule表达式返回是True 那么记录请求链 `controller.Request.Add(resp)`

```
// 单个规则运行
func (controller *PocController) SingleRule(rule *Rule, debug bool) (bool, error) {
	// 格式校验
	err := rule.Verify()
	if err != nil {
		return false, err
	}
	// 替换 set
	rule.ReplaceSet(controller.CEL.ParamMap)
	// 根据原始请求 + rule 生成并发起新的请求 http
	resp, err := controller.DoSingleRuleRequest(rule)
	if err != nil {
		return false, err
	}
	controller.CEL.ParamMap["response"] = resp
	// 匹配search规则
	if rule.Search != "" {
		controller.CEL.ParamMap = rule.ReplaceSearch(resp, controller.CEL.ParamMap)
	}
	// 如果当前rule验证失败，立即释放
	out, err := controller.CEL.Evaluate(rule.Expression)
	if err != nil {
		log.Error("[rule/controller.go:SingleRule cel evaluate error]", err)
		return false, err
	}
	if debug {
		controller.Request.Add(resp)
	} else {
		// 非debug模式下不记录 没有漏洞不记录请求链
		if !out {
			util.ResponsePut(resp)
			return false, nil
		} // 如果成功，记如请求链
		controller.Request.Add(resp)
	}
	return out, err
}
```

controller.DoSingleRuleRequest fasthttp发请求

`poc\rule\controller.go#85`

初始化一个空的req 然后根据 `rule` 的测试位置对应的进行替换进去发起请求返回cel的Response

1. 获取req的fast包
2. `AcquisiteRequest` 从请求池返回一个空请求实例 并将请求内容拷贝到创建的空请求实例
3. 解析获取目录 `curPath`
4. 获取这个插件的影响类型 `Affects`
5. 判断影响级别是哪些再处理
   a. 如果是 params级的 `appendparam` `replaceparam` 将ruile的headers添加到 fastreq包中 通过 `util.DoFasthttpRequest` 发包
   b. 如果是 content级的 通过 `util.DoFasthttpRequest` 发包
   c. 如果是dir级的 目录级漏洞检测 当前路径与 `rule.Path` 拼接出新的路径
   d. 如果是 server级的 赋值 `curPath = rule.Path`
   e. 如果是url级的 这里给注释掉了
6. 为了兼容xray 某些poc没有区分path和query 将url中的 空格和``+ 替换为 `%20` 设置url不进行转义 并替换 `fixedFastReq` 的url
7. 设置 `fixedFastReq` 的 `headers` `method`
8. 如果请求的类型是 `multipart/form-Data` 通过 `util.DealMultipart` 处理body后 更新到 `fixedFastReq`
9. 替换完后发起请求通过 `util.DoFasthttpRequest` 返回 `proto.Response`

```
// 根据原始请求 + rule 生成并发起新的请求
func (controller *PocController) DoSingleRuleRequest(rule *Rule) (*proto.Response, error) {
	fastReq := controller.Request.Fast
	// fixReq : 根据规则对原始请求进行变形
	fixedFastReq := fasthttp.AcquireRequest()
	fastReq.CopyTo(fixedFastReq)
	curPath := string(fixedFastReq.URI().Path())

	affects := controller.Plugin.Affects

	switch affects {
	// param级
	case AffectAppendParameter, AffectReplaceParameter:
		for k, v := range rule.Headers {
			fixedFastReq.Header.Set(k, v)
		}
		return util.DoFasthttpRequest(fixedFastReq, rule.FollowRedirects)
	//	content级
	case AffectContent:
		return util.DoFasthttpRequest(fixedFastReq, rule.FollowRedirects)
	// dir级
	case AffectDirectory:
		// 目录级漏洞检测 判断是否以 "/"结尾
		if curPath != "" && strings.HasSuffix(curPath, "/") {
			// 去掉规则中的的"/" 再拼
			curPath = fmt.Sprint(curPath, strings.TrimPrefix(rule.Path, "/"))
		} else {
			curPath = fmt.Sprint(curPath, "/" ,strings.TrimPrefix(rule.Path, "/"))
		}
	// server级
	case AffectServer:
		curPath = rule.Path
	// url级
	case AffectURL:
		//curPath = curPath, strings.TrimPrefix(rule.Path, "/"))
	default:
	}
	// 兼容xray: 某些 POC 没有区分path和query
	curPath = strings.ReplaceAll(curPath, " ", "%20")
	curPath = strings.ReplaceAll(curPath, "+", "%20")

	fixedFastReq.URI().DisablePathNormalizing= true
	fixedFastReq.URI().Update(curPath)

	for k, v := range rule.Headers {
		fixedFastReq.Header.Set(k, v)
	}
	fixedFastReq.Header.SetMethod(rule.Method)

	// 处理multipart
	contentType := string(fixedFastReq.Header.ContentType())
	if strings.HasPrefix(strings.ToLower(contentType),"multipart/form-Data") && strings.Contains(rule.Body,"\n\n") {
		multipartBody, err := util.DealMultipart(contentType, rule.Body)
		if err != nil {
```

```
                        return nil, err
                }
                fixedFastReq.SetBody([]byte(multipartBody))
        }else {
                fixedFastReq.SetBody([]byte(rule.Body))
        }
        return util.DoFasthttpRequest(fixedFastReq, rule.FollowRedirects)
}
```

raw接口

`scanRoutes.POST("/raw/", scan2.Raw)`

`api\routers\v1\scan\scan\scan.go#92`

1. 会先将raw存成文件
2. 读取raw文件生成req `http.ReadRequest(bufio.NewReader(bytes.NewReader(raw)))`
3. 之后也是创建任务到管道 只是从原来的根据目标生成一个req包变成解析raw转成一个req

```
func Raw(c *gin.Context) {
	scanType := c.PostForm("type")
	vulList := c.PostFormArray("vul_list")
	remarks := c.PostForm("remarks")

	if scanType != "multi" && scanType != "all" {
		c.JSON(msg.ErrResp("扫描类型为multi或all"))
		return
	}
	target, err := c.FormFile("target")
	if err != nil {
		c.JSON(msg.ErrResp("文件上传失败"))
		return
	}
	// 存文件
	filePath := file.UploadTargetsPath(path.Ext(target.Filename))
	err = c.SaveUploadedFile(target, filePath)

	oreq, err := http.ReadRequest(bufio.NewReader(bytes.NewReader(raw)))
	if err != nil || oreq == nil {
		c.JSON(msg.ErrResp("生成原始请求失败"))
		return
	}
	if !oreq.URL.IsAbs() {
		scheme := "http"
		oreq.URL.Scheme = scheme
		oreq.URL.Host = oreq.Host
	}

	plugins, err := rule.LoadDbPlugin(scanType, vulList)
	if err != nil || plugins == nil {
		c.JSON(msg.ErrResp("插件加载失败" + err.Error()))
		return
	}

	oReqUrl := oreq.URL.String()

	token := c.Request.Header.Get("Authorization")
	claims, _ := util.ParseToken(token)

	task := db.Task{
		Operator: claims.Username,
		Remarks: remarks,
		Target:  oReqUrl,
	}
	db.AddTask(&task)
	taskItem := &rule.TaskItem{
		OriginalReq: oreq,
		Plugins:     plugins,
		Task:        &task,
	}

	c.JSON(msg.SuccessResp("任务下发成功"))
	go rule.TaskProducer(taskItem)
```

```
        go rule.TaskConsumer()
        return
}
```

List

```
scanRoutes.POST("/list/", scan2.List)
```

```
api\routers\v1\scan\scan\scan.go#175
```

将目标文件存下来 解析遍历创建多个task任务

后续也是一样的创建的任务到管道

```
func List(c *gin.Context) {
        scanType := c.PostForm("type")
        vulList := c.PostFormArray("vul_list")
        remarks := c.PostForm("remarks")

        if scanType != "multi" && scanType != "all" {
                c.JSON(msg.ErrResp("扫描类型为multi或all"))
                return
        }

        target, err := c.FormFile("target")
        if err != nil {
                c.JSON(msg.ErrResp("文件上传失败"))
                return
        }
        // 存文件
        filePath := file.UploadTargetsPath(path.Ext(target.Filename))
        err = c.SaveUploadedFile(target, filePath)

        if err != nil || !file.Exists(filePath) {
                c.JSON(msg.ErrResp("文件保存失败"))
                return
        }

        // 加载poc
        plugins, err := rule.LoadDbPlugin(scanType, vulList)
        if err != nil{
                c.JSON(msg.ErrResp("插件加载失败" + err.Error()))
                return
        }
        if len(plugins) == 0 {
                c.JSON(msg.ErrResp("插件加载失败" + err.Error()))
                return
        }
        targets := file.ReadingLines(filePath)

        token := c.Request.Header.Get("Authorization")
        claims, _ := util.ParseToken(token)

        var oReqList []*http.Request
        var taskList []*db.Task

        for _, url := range targets {
                oreq, err := util.GenOriginalReq(url)
                if err != nil {
                        continue
                }
                task := db.Task{
                        Operator: claims.Username,
                        Remarks: remarks,
                        Target:  url,
                }
                db.AddTask(&task)
```

```
                oReqList = append(oReqList, oreq)
                taskList = append(taskList, &task)
        }

        if len(oReqList) == 0 || len(taskList) ==0 {
                c.JSON(msg.ErrResp("url列表加载失败"))
                return
        }
        c.JSON(msg.SuccessResp("任务下发成功"))

        for index, oreq := range oReqList {
                taskItem := &rule.TaskItem{
                        OriginalReq: oreq,
                        Plugins:     plugins,
                        Task:        taskList[index],
                }
                go rule.TaskProducer(taskItem)
                go rule.TaskConsumer()
        }
        return
}
```

对于cel-go的使用理解

这里抄了 pocassist的cel-go的代码 加上自己的注释理解 写了一个demo
简单对cel-go进行一个使用

```
/*
 * @Date: 2022-03-15 16:20:30
 * @LastEditors: recar
 * @LastEditTime: 2022-03-16 10:29:56
 */

package main

import (
        "crypto/md5"
        "fmt"
        "math/rand"
        "strings"

        "cel/proto"

        "github.com/google/cel-go/cel"
        "github.com/google/cel-go/checker/decls"
        "github.com/google/cel-go/common/types"
        "github.com/google/cel-go/common/types/ref"
        "github.com/google/cel-go/interpreter/functions"
        exprpb "google.golang.org/genproto/googleapis/api/expr/v1alpha1"
)

// 定义一个自定义函数
// 判断s1是否包含s2, 忽略大小写
// 描述
var iContainsDec = decls.NewFunction("icontains", decls.NewInstanceOverload("string_icontains_string", []*exprpb.Type{dec

// 实现
// 这里的Operator 是运算符 即 字符串.icontains(字符串)
// Binary 是定义这个函数 通过ref反射动态执行 先判断类型是否是字符串 然后再执行 strings.Contains
var iContainsFunc = &functions.Overload{
        Operator: "string_icontains_string",
        Binary: func(lhs ref.Val, rhs ref.Val) ref.Val {
                v1, ok := lhs.(types.String)
                if !ok {
                        return types.ValOrErr(lhs, "unexpected type '%v' passed to icontains", lhs.Type())
                }
                v2, ok := rhs.(types.String)
                if !ok {
                        return types.ValOrErr(rhs, "unexpected type '%v' passed to icontains", rhs.Type())
                }
                return types.Bool(strings.Contains(strings.ToLower(string(v1)), strings.ToLower(string(v2))))
        },
}

// 自定义函数 randomInt
var randomIntDec = decls.NewFunction("randomInt", decls.NewOverload("randomInt_int_int", []*exprpb.Type{decls.Int, decls.
var randomIntFunc = &functions.Overload{
        Operator: "randomInt_int_int",
        Binary: func(lhs ref.Val, rhs ref.Val) ref.Val {
                from, ok := lhs.(types.Int)
                if !ok {
```

```
                        return types.ValOrErr(lhs, "unexpected type '%v' passed to randomInt", lhs.Type())
                }
                to, ok := rhs.(types.Int)
                if !ok {
                        return types.ValOrErr(rhs, "unexpected type '%v' passed to randomInt", rhs.Type())
                }
                min, max := int(from), int(to)
                return types.Int(rand.Intn(max-min) + min)
        },
}

//  字符串的 md5
var md5Dec = decls.NewFunction("md5", decls.NewOverload("md5_string", []*exprpb.Type{decls.String}, decls.String))
var md5Func = &functions.Overload{
        Operator: "md5_string",
        Unary: func(value ref.Val) ref.Val {
                v, ok := value.(types.String)
                if !ok {
                        return types.ValOrErr(value, "unexpected type '%v' passed to md5_string", value.Type())
                }
                return types.String(fmt.Sprintf("%x", md5.Sum([]byte(v))))
        },
}

// 初始化 cel的环境变量即自定义的函数和变量
// 是 Library接口的实现
/*
type Library interface {
        // CompileOptions returns a collection of funcitional options for configuring the Parse / Check
        // environment.
        CompileOptions() []EnvOption

        // ProgramOptions returns a collection of functional options which should be included in every
        // Program generated from the Env.Program() call.
        ProgramOptions() []ProgramOption
}
*/
type CustomLib struct {
        // 声明
        envOptions []cel.EnvOption
        // 实现
        programOptions []cel.ProgramOption
}

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

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

// 第一步定义 cel options
func InitCelOptions() CustomLib {
        custom := CustomLib{}
```

```
	custom.envOptions = []cel.EnvOption{
		cel.Container("proto"),
		//	注入一种类型
		cel.Types(
			&proto.UrlType{},
			&proto.Request{},
			&proto.Response{},
		),
		// 定义变量
		cel.Declarations(
			decls.NewVar("request", decls.NewObjectType("pkg.proto.Request")),
			decls.NewVar("response", decls.NewObjectType("pkg.proto.Response")),
		),
		// 定义函数
		cel.Declarations(iContainsDec, randomIntDec, md5Dec),
	}
	// 实现的函数
	custom.programOptions = []cel.ProgramOption{cel.Functions(iContainsFunc, randomIntFunc, md5Func)}
	return custom
}

// 第二步 根据cel options 创建 cel环境
func InitCelEnv(c *CustomLib) (*cel.Env, error) {
	// cel.Lib 的参数是Library 是 CustomLib 的接口
	return cel.NewEnv(cel.Lib(c))
}

//	如果有set：追加set变量到 cel options
// 这里的set 就是yaml的set 定义的一些变量
func (c *CustomLib) AddRuleSetOptions(args []map[string]string) {
	for _, arg := range args {
		// 在执行之前是不知道变量的类型的，所以统一声明为字符型
		// 所以randomInt虽然返回的是int型，在运算中却被当作字符型进行计算，需要重载string_*_string
		for k := range arg {
			v := arg[k]
			var d *exprpb.Decl
			// 下面设置了这三种字符串设置类型
			if strings.HasPrefix(v, "randomInt") {
				d = decls.NewVar(k, decls.Int)
			} else if strings.HasPrefix(v, "newReverse") {
				d = decls.NewVar(k, decls.NewObjectType("proto.Reverse"))
			} else {
				d = decls.NewVar(k, decls.String)
			}
			// 追加到 envOpt中
			c.envOptions = append(c.envOptions, cel.Declarations(d))
		}

	}
}

//	计算单个表达式
func Evaluate(env *cel.Env, expression string, params map[string]interface{}) (ref.Val, error) {
	ast, iss := env.Compile(expression)
	if iss.Err() != nil {
```

```
                return nil, iss.Err()
        }
        prg, err := env.Program(ast)
        if err != nil {
                return nil, err
        }
        out, _, err := prg.Eval(params)
        if err != nil {
                return nil, err
        }
        return out, nil
}

func main() {
        //      1.生成cel env环境
        option := InitCelOptions()
        //      动态添加变量到env里 这里即对poc里的set
        set := []map[string]string{}
        rad := map[string]string{"rad": "randomInt(1,10)"}
        set = append(set, rad)
        option.AddRuleSetOptions(set)
        env, err := InitCelEnv(&option)
        if err != nil {
                fmt.Println("[rule/cel.go:Init init cel env error]", err)
        }
        // icontains
        expression1 := `"aaa".icontains("aaa")`
        params := make(map[string]interface{})
        out, err := Evaluate(env, expression1, params)
        if err != nil {
                fmt.Println(err)
        }
        fmt.Printf("expression1: %v\n", out)
        // set
        setMap := make(map[string]interface{})
        for _, s := range set {
                for k := range s {
                        expression2 := s[k]
                        out, err = Evaluate(env, expression2, params)
                        if err != nil {
                                fmt.Println(err)
                        }
                        setMap[k] = out.Value()
                }
        }
        //print set变量

        for k, v := range setMap {
                fmt.Printf("expression2 set k: %s v: %d\n", k, v)
                // 将set的变量添加的env里
                params[k] = v
        }
        // 把set替换request中的body或者headers
        // eval 表达式 匹配关键字返回true或者false
```

```
	expression3 := `string(rad)`
	out, err = Evaluate(env, expression3, params)
	if err != nil {
		fmt.Println(err)
	}
	fmt.Printf("expression3: %s\n", out)

	// 自定义一个变量如下
	params["recar"] = "cel-go"
	expression4 := `recar`
	// 定义添加变量的类型
	recar := decls.NewVar("recar", decls.String)
	option.envOptions = append(option.envOptions, cel.Declarations(recar))
	env, err = InitCelEnv(&option)
	if err != nil {
		fmt.Println("[rule/cel.go:Init init cel env error]", err)
	}
	out, err = Evaluate(env, expression4, params)
	if err != nil {
		fmt.Println(err)
	}
	fmt.Printf("expression4: %s\n", out)

}
```

参考

cel-go https://codelabs.developers.google.com/codelabs/cel-go#0

pocassist https://pocassist.jweny.top/