# SecretDetection

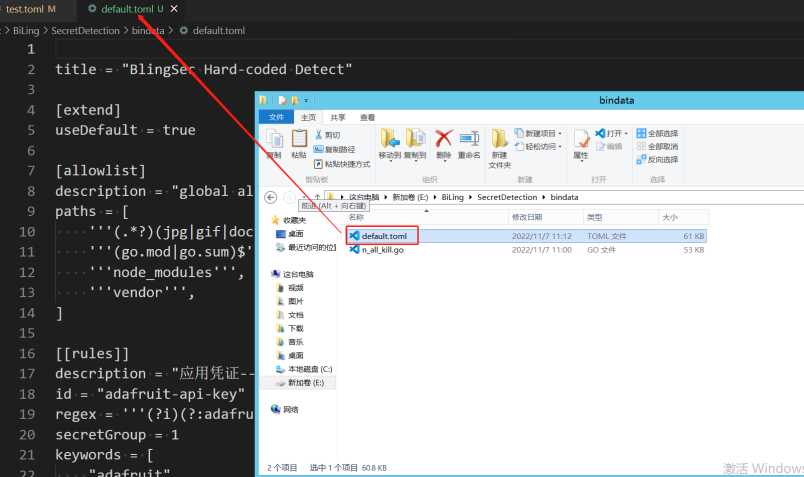
# 简介

SecretDetection是使用golang写的进行硬编码检测的二进制程序，程序可以进行全平台快速编译，速度快，并发高，程序体积小，运行稳定，可以通过命令行方式对目标目录进行凭证扫描。

# 编译方法

## 编辑自定义规则（阅读时如对规则无改动需求可暂时跳过此部分）

SecretDetection/bindata/default.toml 是程序默认的规则文件，如果要修改程序的默认规则文件（而非通过-c命令进行外部加载），可修改此处的default.toml文件来编辑程序的默认加载规则，default.toml文件如下图所示:



default.toml是程序默认的加载规则，编译之前在此处编辑规则文件即可。这是toml格式的文件。

如果希望在默认规则基础上进行修改，通常修改[allowlist]或者添加[[rules]]（至于问为什么rule左右两边的括号是两个，是因为[[rules]]表示rules是个列表）这两处。

关于该文件中的其他的配置，主要是程序运行时采用的一些默认配置以及熵的阈值，随意修改可能影响程序稳定性，不建议进行修改。

### [allowlist]非扫描对象的后缀名黑名单

如果要修改[allowlist]，可以仿照原文示例对原文进行删减。[allowlist]主要是解决扫描时根据后缀名对哪些文件进行跳过。比如要进行硬编码扫描，我们希望程序扫描.go、.java及.c等代码程序，但是我们通常不希望对配置文件、数据文件等进行扫描，如java项目中的.properties文件、config.txt、pom.xml文件等配置文件，或者作为数据库初始化或者备份的文件，如.sql结尾的文件。

如果不希望扫描这些配置项，可根据扫描目标的情况在下面的allowlist的paths列表中的后缀名进行添加；如果希望扫描，则在该处配置项中删去特定后缀名。allowlist配置项原文如下：

[allowlist]

description = "global allow lists"

paths = [

'''(.\*?)(jpg|gif|doc|docx|zip|xls|pdf|bin|svg|socket|csv|xlsx|ini|properties|proterty|csr|cer|der|p12|pem|zip|ppt|pptx|exe|toml|yaml|txt|m|xml|plist|config|cfg|sql|bak|data|log|json|sqlite|db|conf)$''',

'''(go.mod|go.sum)$''',

'''node\_modules''',

'''vendor''',

]

### [[rules]]匹配规则

匹配规则主要是[[rules]]，根据扫描目标情况，模仿toml文件中的[[rules]]示例增加扫描规则即可。

示例如下：

[[rules]]

description = "应用凭证--Adafruit API Key"

id = "adafruit-api-key"

regex = '''(?i)(?:adafruit)(?:[0-9a-z\-\_\t .]{0,20})(?:[\s|']|[\s|"]){0,3}(?:=|>|:=|\|\|:|<=|=>|:)(?:'|\"|\s|=|\x60){0,5}([a-z0-9\_-]{32})(?:['|\"|\n|\r|\s|\x60|;]|$)'''

secretGroup = 1

keywords = [

"adafruit",

]

description表示描述，id表示规则的ID，regex表示正则规则，keywords表示该凭证规则在代码中可能会出现什么样的关键字，通常是对应凭证的类型，如微信的API，可能出现的名字如“wx”等，可以写多个关键字。

关于regex中的正则表达式需要多花几分钟了解一下该配置文件中的表达式规则：

(?i) 标识大小写不敏感，相反如 (?-i:x)——x 区分大小写（大小写敏感）

(?:key|api|token) 这个(?:)的意思就是，不保存捕获分组，模仿程序即可。

\b不匹配任何东西，只是标识\w和\W之间的位置

\x60 标识 `

这里注意，比如自行编写的规则 regex = '''(?i)\bw((?i)[a-z0-9\_]{35})(?:['|\"|\n|\r|\s|\x60]|$)'''

这里的第二个((?i)xxxxx)，这里面捕获的数据，是要返回结果作为匹配到的凭证（secret），这里不能不捕获东西。正则表达式整体能匹配到的内容会返回结果作为Match中的值。

千万要注意，只能让密钥匹配那里给捕获，这里加规则的时候，如果使用到()进行匹配，不要忘记写作(?:xXXX)

这里举个示例，比如我想增加企业微信的API的[[rules]]，示例如下：

# CxydMDAEscOL7Hqbqphf77rY3HpVsJDu4cizL

[[rules]]

description = "Qiye WeiXin Developer App Secret"

id = "qiye-weixin-developer-app-secret"

regex = '''(?i)\b((?i)[a-z0-9]{37})(?:['|\"|\n|\r|\s|\x60]|$)'''

secretGroup = 1

keywords = [

"qyapi",

"weixin",

]

这里再举个示例，比如我想写一个通用型、具备高检出能力（同时误报率也非常高）的规则，示例如下：

[[rules]]

description = "高检出模糊匹配--模糊匹配"

id = "通用关键字匹配"

regex = '''(?i)(?:['|\"|:|=]{1,3})((?i)[\w]{6,70})(?:['|\"|\n|\r|\s|\x60]|$)'''

secretGroup = 1

keywords = [

"access\_key","access\_token","admin\_pass","admin\_user","algolia\_admin\_key","algolia\_api\_key","alias\_pass","alicloud\_access\_key","amazon\_secret\_access\_key","amazonaws","ansible\_vault\_password","aos\_key","api\_key","api\_key\_secret","api\_key\_sid","api\_secret","api.googlemaps AIza","apidocs","apikey","apiSecret","app\_debug","app\_id","app\_key","app\_log\_level","app\_secret","appkey","appkeysecret","application\_key","appsecret","appspot","auth\_token","authorizationToken","authsecret","aws\_access","aws\_access\_key\_id","aws\_bucket","aws\_key","aws\_secret","aws\_secret\_key","aws\_token","AWSSecretKey","b2\_app\_key","bashrc password","bintray\_apikey","bintray\_gpg\_password","bintray\_key","bintraykey","bluemix\_api\_key","bluemix\_pass","browserstack\_access\_key","bucket\_password","bucketeer\_aws\_access\_key\_id","bucketeer\_aws\_secret\_access\_key","built\_branch\_deploy\_key","bx\_password","cache\_driver","cache\_s3\_secret\_key","cattle\_access\_key","cattle\_secret\_key","certificate\_password","ci\_deploy\_password","client\_secret","client\_zpk\_secret\_key","clojars\_password","cloud\_api\_key","cloud\_watch\_aws\_access\_key","cloudant\_password","cloudflare\_api\_key","cloudflare\_auth\_key","cloudinary\_api\_secret","cloudinary\_name","codecov\_token","config","conn.login","connectionstring","consumer\_key","consumer\_secret","credentials","cypress\_record\_key","database\_password","database\_schema\_test","datadog\_api\_key","datadog\_app\_key","db\_password","db\_server","db\_username","dbpasswd","dbpassword","dbuser","deploy\_password","digitalocean\_ssh\_key\_body","digitalocean\_ssh\_key\_ids","docker\_hub\_password","docker\_key","docker\_pass","docker\_passwd","docker\_password","dockerhub\_password","dockerhubpassword","dot-files","dotfiles","droplet\_travis\_password","dynamoaccesskeyid","dynamosecretaccesskey","elastica\_host","elastica\_port","elasticsearch\_password","encryption\_key","encryption\_password","env.heroku\_api\_key","env.sonatype\_password","eureka.awssecretkey",

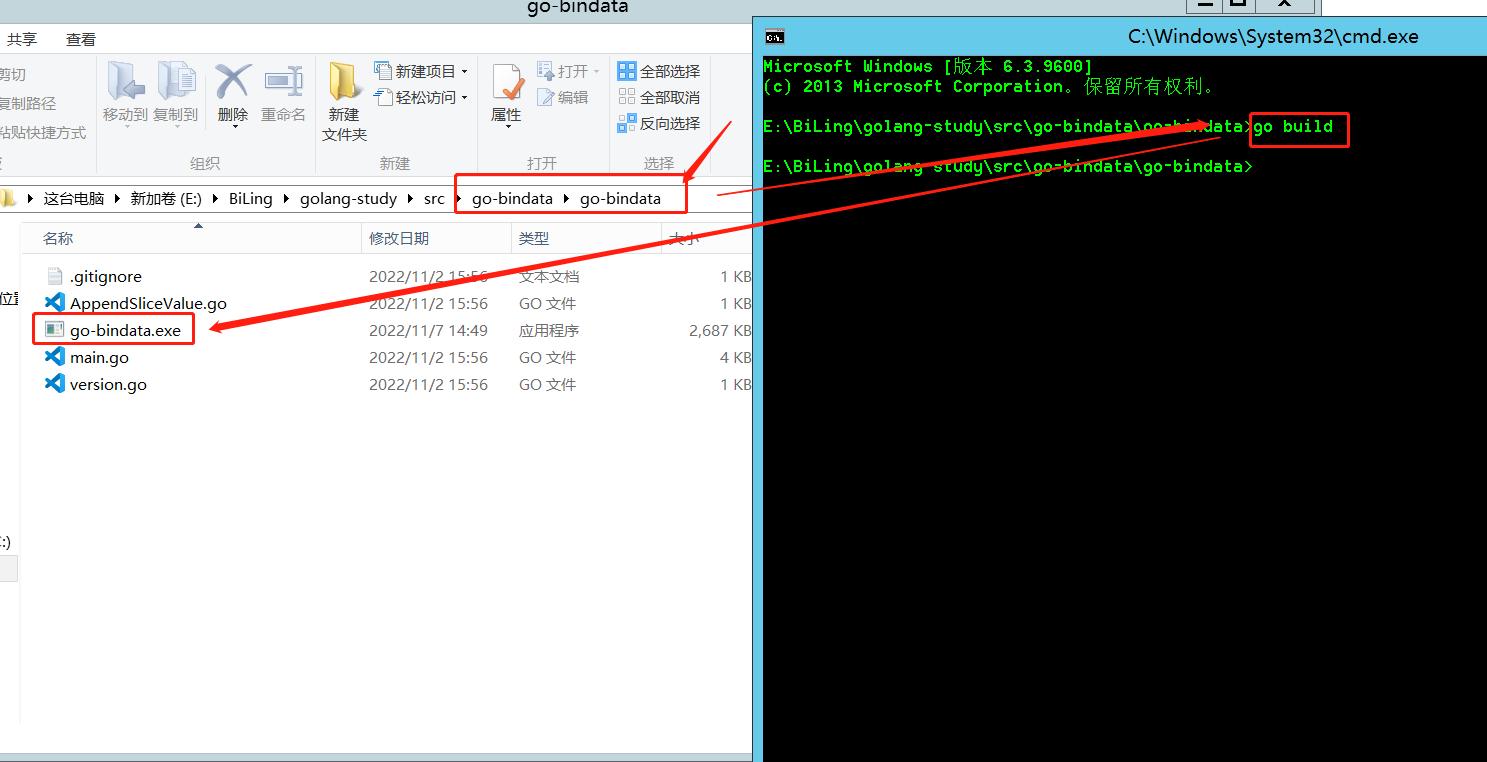
]

该示例规则基本可以匹配到绝大多数的凭证，但是误报率也相当之高。

## 编译规则文件

电脑上需要先准备有bindata工具：

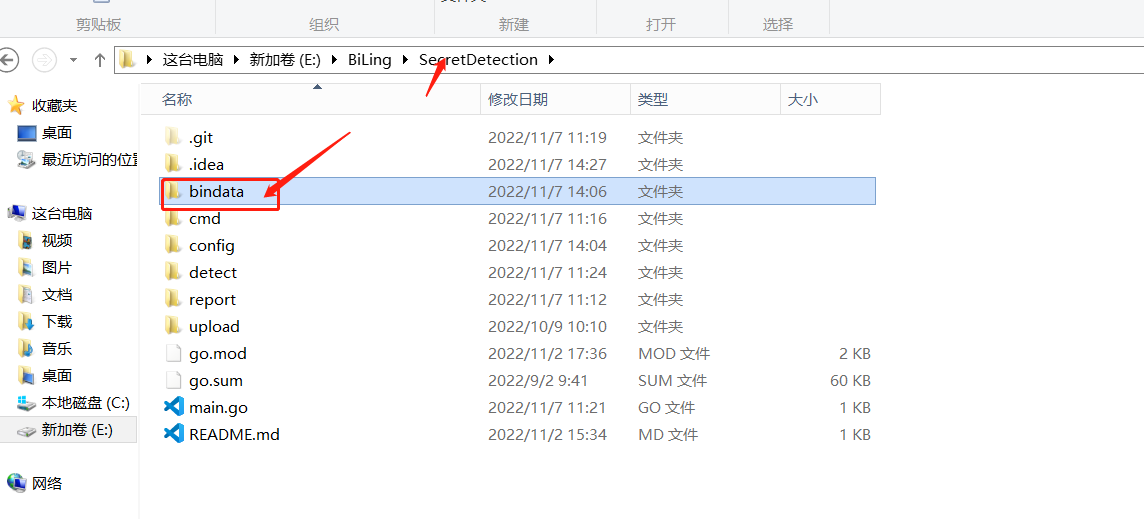
需要访问https://github.com/go-bindata/go-bindata网址，将该仓库克隆到本地，然后切入go-bindata目录，通过go build命令编译出go-bindata.exe程序，如图所示：



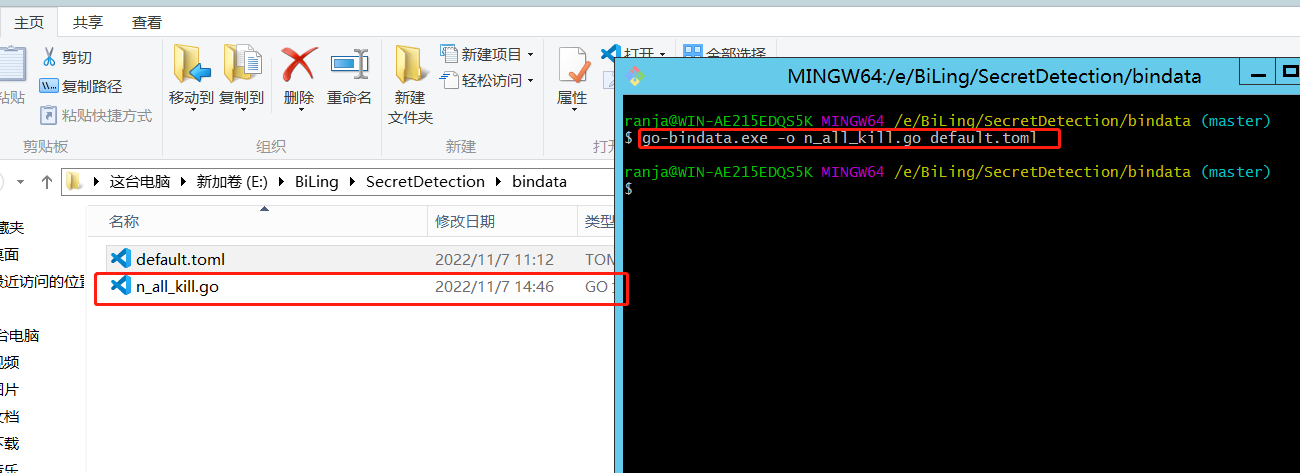
建议将该编译好的程序的目录添加到环境变量中，方便后面对toml配置文件的编译。

接下来开始编译toml文件：

进入SecretDetection/bindata/目录

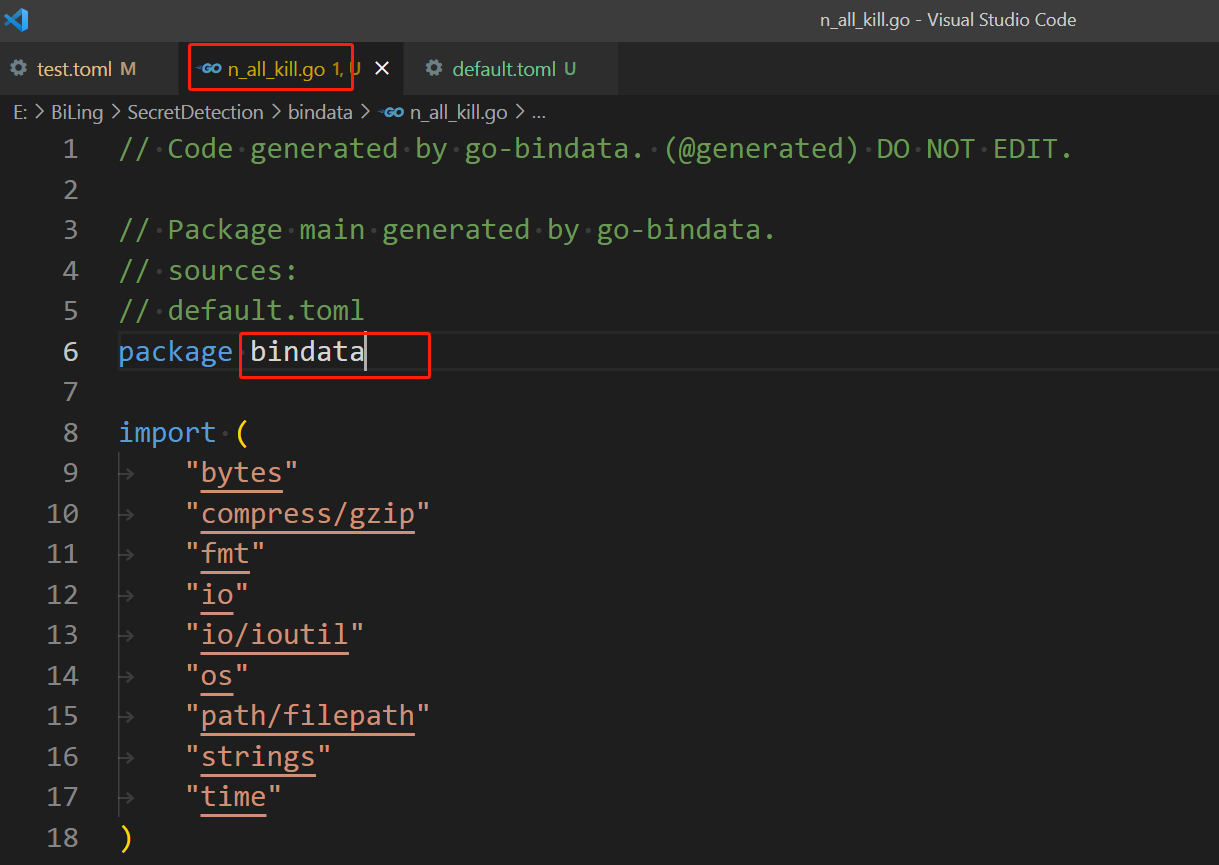


执行命令: go-bindata.exe -o n\_all\_kill.go default.toml,将默认的规则文件default.go编译为n\_all\_kill.go文件。



可以看到输出n\_all\_kill.go文件。

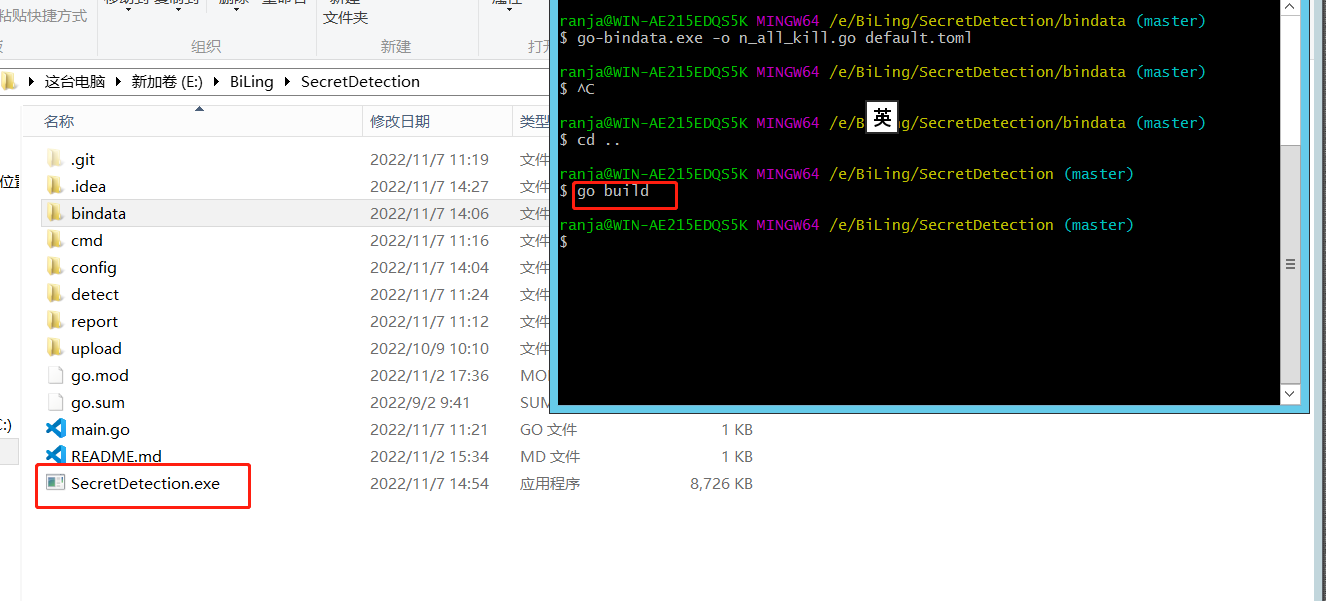
然后使用编辑器打开n\_all\_kill.go文件修改下包名，即将package main修改为package bindata,如图：



规则文件就算编译完毕了。

## 编译程序

切到SecretDetection目录，执行 go build 命令，获得编译后的可执行文件：



# 使用方法

## 帮助信息

我们通常执行SecretDetection.exe detect命令，-h可以查看程序的帮助信息

SecretDetection.exe -h



SecretDetection.exe detect -h



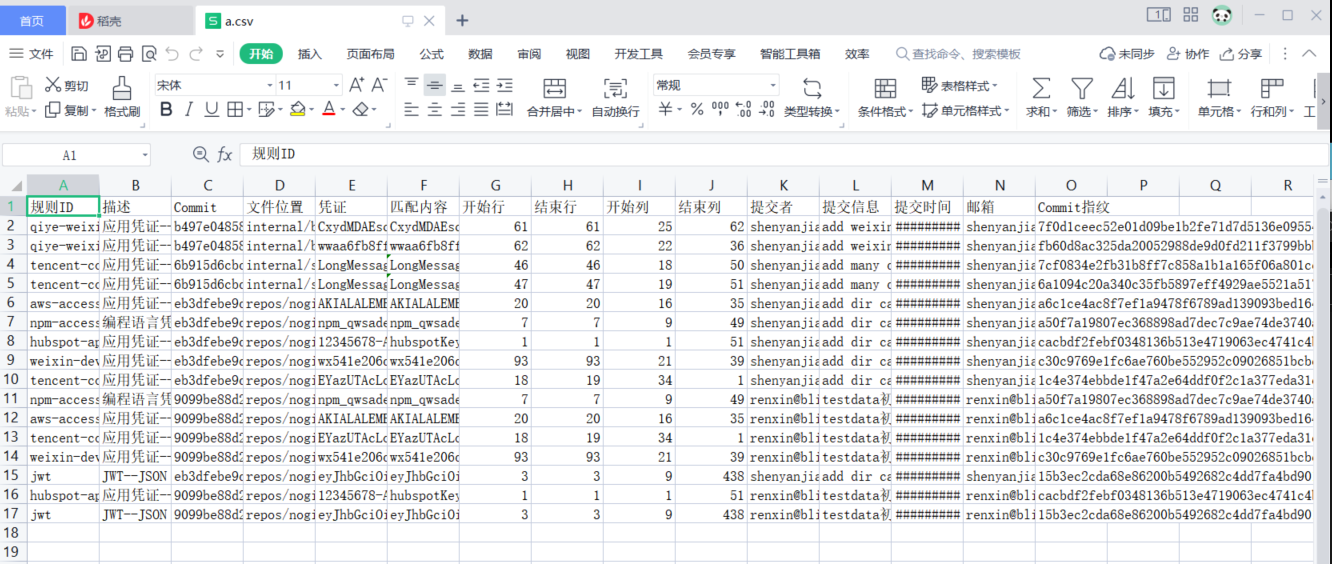
## 扫描并保存结果

比如，我们扫描特定目录，并将结果以csv格式保存在a.csv文件中。

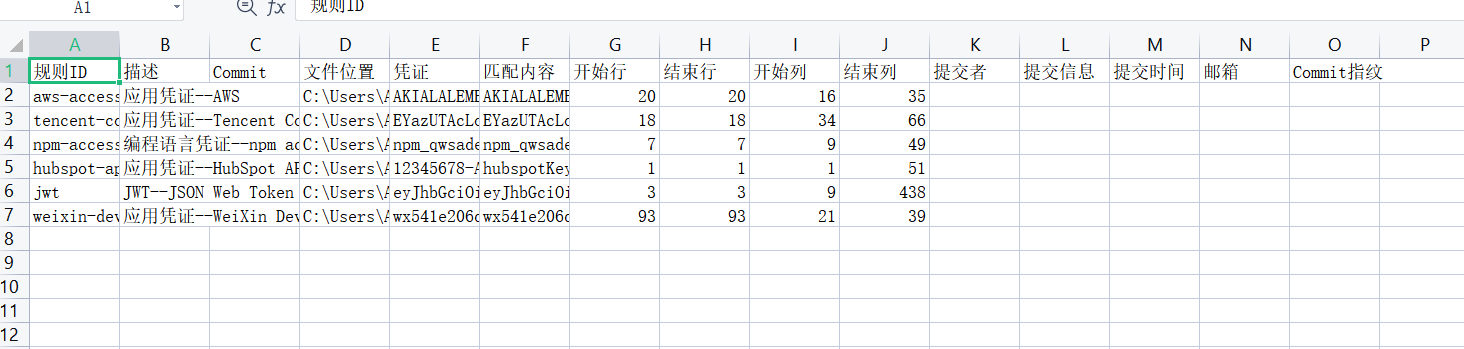
运行命令:SecretDetection.exe detect -s C:\Users\ranja\Videos\testdata -v -r a.csv -f csv

我们查看a.csv，如图：

如果目标目录中有.git目录，则会有git log等提交者、commit等相关信息：

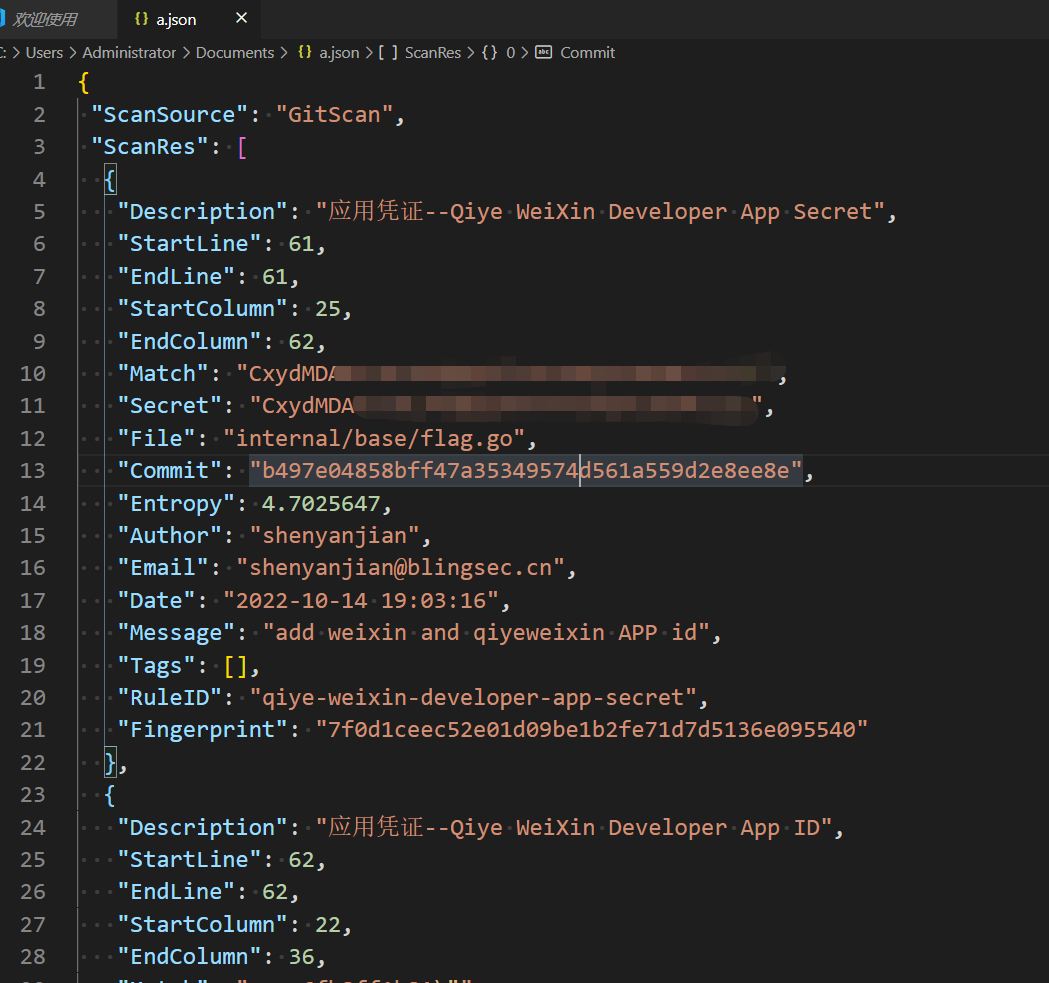


如果目标目录不存在.git目录，则不会有相关信息，如下：



除了csv格式，也可以将结果以json格式保存。

执行命令: SecretDetection.exe detect -s C:\Users\ranja\Videos\testdata -r a.json -f json



## 扫描git仓库或者非git仓库以及主机缺失git命令处理情况

给程序指定 detect -s 参数即可进行扫描特定目录，如SecretDetection.exe detect -s 目标目录。

如果目标目录下存在.git目录（即git仓库），则程序会运行git log和git diff命令获取该仓库的git信息，如commit记录、不同文件的提交人等信息，如上图中可以看到某个文件由某个人提交这种信息。此过程中的程序不会连接git地址，仅仅是本地运行git log和git diff命令，不会产生任何流量。

但是获取上述信息，需要主机上提前安装有git，并有git命令。否则会运行失败。

对于这种情况，SecretDetection.exe扫描的目标目录下如果没有.git目录（即非git仓库），则程序不会调用主机的git命令。

对于主机没有git命令，但是还要扫描本地git仓库的情况，可以删除掉.git目录再使用SecretDetection命令进行扫描，这样SecretDetection会进行不调用git命令的凭证扫描。

# 外部加载自定义规则

在使用 -h 参数查看帮助信息时，并看不到存在 -c 参数的相关说明，这是因为隐藏了--config,-c的帮助信息的显示。



程序除了可以直接修改规则default.toml并编译以外，也可以让程序外部加载用户自定义的toml文件并执行。这点主要考虑到用户可能自定义不扫描文件的黑名单后缀，或者根据自己的代码项目情况配置自己的API、密钥等新的规则。

以下以test.toml为用户自定义规则文件的命名示例进行演示。

用户新建test.toml的空白文件，开始按照toml格式进行编辑。用户可以自定义[allowlist]或者[[rules]]，两者并非必须并存。用户可以单添加自己的[[rules]]，或者单写[allowlist]，或者用户自定义的[allowlist]和[[rules]]两者都存在。

关于配置文件中的规则如何编辑，请参考本文“编译方法”中的“编辑自定义规则”。

默认的规则和用户外部加载规则的关系如下：

用户添加[[rules]]，如果该条规则的id是默认规则(default.toml)之外的id，对于程序而言，该规则会添加在程序运行时的[[rules]]中；如果id和默认规则的某条一样，对于程序而言，相当于重新改写了默认规则中的该条规则。

用户添加[allowlist]，只需要写paths，不需要写description。用户的paths会直接覆盖程序运行时的[allowlist]的paths。

简单来说，用户外部自定义的规则中的[[rules]]可以理解为“添加”，而[allowlist]是“直接替换”。这么做主要是为了用户可以根据自己的检测目标代码的需求，设置我们程序不曾考虑过的凭证的规则；用户自定义allowlist可以更灵活的按照用户想法自行写后缀名黑名单，灵活的检测不同的后缀文件。

# 注意

程序为了提高运行效率，默认情况下，在对git仓库进行扫描时会跳过大于2MB的文件，对非git仓库进行扫描时会跳过大于60000行的文件。通常该类文件非代码文件，更可能的是数据文件。通常情况下跳过此类文件的扫描，可以让程序运行更加稳定和快速。

极少情况下，如果用户有特殊需求，需要扫描此类大文件，请联系我们的工程师调整文件大小的限制阈值。

# 相关术语

## 凭证

计算机领域中，凭证通常指程序如何访问、连接其他服务或者网络，比如连接数据库需要设置数据库的账号密码，才能正常访问到数据库；访问一个在线服务的API，通常会要求API Key或者Secret；将代码通过git上传到私有的git服务器，也需要携带git生成的token值，才能连接到git的私有仓库。

所以，账户名、密码、API地址、访问API需要携带的token、session和secret都是凭证，除此以外，用于进行加密、解密算法中的密钥、公钥、私钥通常也被认为是凭证信息。

## 硬编码

硬编码是将数据直接嵌入到程序或其他可执行对象的源代码中的软件开发实践，而不是从外部源获取数据或在运行时生成数据。

硬编码数据通常只能通过编辑源代码并重新编译可执行文件来修改。硬编码的数据最适合用于不变的信息，例如物理常数、版本号和静态文本元素。

软编码数据通过用户输入、文本文件、INI 文件、HTTP 服务器响应、配置文件、预处理器宏、外部常量、数据库、命令行参数对任意信息进行编码，并在运行时确定。

硬编码给计算机工程师们带来了相当大的痛苦，一个问题是更改配置信息需要更改代码，然后重新编译再部署系统；另一个问题是配置被硬编码到系统的多个位置，程序员必须记住硬编码存在的所有地方，这可能会导致很大的维护问题。

为了避免将配置信息硬编码到系统中，我们经常将配置移至外部源，即文本文件或数据库表。这允许我们在不重构系统的情况下更改配置。

该处可参考CVE-2016-4437，未更改默认密钥的Apache Shiro服务容易受到反序列化任意命令执行的工具，但是管理员可以通过更改密钥，直接避免shiro服务的此类被针对的漏洞。这得益于apache shiro的开发工程师没有将密钥直接硬编码到程序中，因此服务启动后的任意运维人员都可以简单的通过更改密钥，避免默认密钥泄露导致的易受攻击的问题。

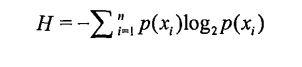
## 密码密钥硬编码的检查

### 鉴别密码密钥方法

#### 香农熵(Shannon entropy)

密钥的长度决定了密钥空间，通常以位为单位。密钥空间越大，密钥被攻破的难度就越大。

密钥是由密钥空间的随机值构成。对于任意一个随机变量 X，它的熵定义如下：



变量的不确定性越大，熵也就越大，把它搞清楚所需要的信息量也就越大。

同等长度的字符串，通常密钥的熵值更高。而密钥为避免彩虹攻击，在取值上会更加的离散，会尽量采用不重复的字符。就像我们为了增加密码的复杂性，要求长度不小于8，必须包含大小写、特殊字符、以及数字一样的道理，所以密钥的熵值会比一般的文本要高的多。我们就是利用这点来识别字符串是否是密钥，减少非密钥误报率。

#### 文本内容规则匹配

在使用熵进行计算的同时，凭证检测程序也会对内容文本进行分析，避免熵检测带来的凭证硬编码漏报的情况。

凭证检测程序在读取本地的代码仓库时，使用git diff及git log命令查找目前仓库中所有已存在文件的上传作者及 commit id，然后开始对文件进行匹配和分析。

SecretDetection中配置了大量的API、密钥、证书等相关的匹配规则，可以准确识别出主流云服务、大多数主流厂商的访问API及secret格式；除此以外，开发人员在测试过程中可能无意注释了一些敏感信息然后遗忘的话，程序也会通过匹配的方式记录此类问题。

在分析完毕后，会记录全部的存在硬编码的可疑风险，并附上代码位置、作者、commit id等信息。