DEC 29, 2017 - 20 MINUTE READ - COMMENTS - GO

Simple SSH Harvester in Go

During my Go SSH adventures at Hacking with Go I wanted to write a simple SSH harvester. As usual, the tool turned out to be much larger than I thought.

I realized I cannot find any examples of SSH certificate verification. There are a few examples for host keys here and there. Even the `certs_test.go` file just checks the host name. There was a typo in an error message[1] in the `crypto/ssh` package but I think because this is not very much used, had gone unreported.

Here's my step by step guide to writing this tool by piggybacking on SSH host verification callbacks. Hopefully this will make it easier for the next person.

You can find the code here:

- https://github.com/parsiya/SSH-Scanner/blob/master/SSHHarvesterv1.go

TL;DR: verifying SSH servers

1. Create an instance of ssh.CertChecker.
2. Set callback functions for `IsHostAuthority`, `IsRevoked` and optionally `HostKeyFallback`.
   - `IsHostAuthority`'s callback should return `true` for valid certificates.
   - `IsRevoked`'s callback should return `false` for valid certificates.
   - `HostKeyFallback`'s callback should return `nil` for valid certificates.
3. Create an instance of ssh.ClientConfig.
4. Set `HostKeyCallback` in `ClientConfig` to `&ssh.CertChecker.CheckHostKey`.
5. CheckHostKey will verify the certificate based on other callback functions.
6. The certificate can be accessed in `IsRevoked` callback function.

Go to `Parsing SSH certificates` to skip the fodder.

Table of Contents

Before we start

1. Think of this as a simple Proof of Concept (PoC). I will keep this version in the clone. However, I will keep building upon this to make it a full-blown SSH vuln scanner using non-standard libraries.
2. I kept to standard libraries. For example I know there are better CLI managers than flag out there like Cobra and CLI.

3. Everything is in one big file, this will hopefully be fixed in the vuln scanner.

Code analysis

I am not completely trying to deflect criticism but security scripts are a different beast. You want to write something that does some specific thing and alerts you the moment it stops working so you can fix/redo. That said, please let me know if there are any huge errors or if I can do something much better.

Constants and usage

We can either pass a file with `-in`. The file should have one address on each line:

Input file example

```
127.0.0.1:22
[2001:db8::68]:1234
```

Or we can pass addresses with `-t` separated by commas:

- `SSHHarvester.exe -t 127.0.0.1:22,[2001:db8::68]:1234`

Output file is specified with `-out`.

Constants - Usage

```
const (
    mUsage = "SSH Harvester gathers and publishes info about SSH servers.\n" +
        "Addresses should be in format of 'host:port'.\n" +
        "Input file should have one address on each line " +
        "and addresses provided to -targets should be separated by commas.\n" +
        "-in and -targets are mutually exclusive, use one.\n" +
        "Examples:\n" +
        "go run SSHHarvester1.go -t 127.0.0.1:12334,192.168.0.10:22\n" +
        "go run SSHHarvester1.go -i inputfile.txt\n" +
        "go run SSHHarvester1.go -i inputfile.txt -out output.txt\n"
    outUsage = "output report file"
    inUsage  = "input file"
    tUsage   = "addresses separated by comma"
    vUsage   = "print extra info"

    // Delimiter for host:port
    AddressDelim = ":"
    // // Delimiter for IPv6 addresses
    // IPv6Delim = "[]"

    // Log prefix - note the trailing space
    LogPrefix = "[*] "

    // Test SSH username/password - not really important
    TestUser     = "user"
    TestPassword = "password"

    // Timeout in seconds
    Timeout = 5 * time.Second
)

// Usage string
func usage() {
    usg := mUsage
    usg += fmt.Sprintf("\n  -i, -in\tstring\t%s", inUsage)
    usg += fmt.Sprintf("\n  -o, -out\tstring\t%s", outUsage)
    usg += fmt.Sprintf("\n  -t, -targets\tstring\t%s", tUsage)
    usg += fmt.Sprintf("\n  -v, -verbose\tstring\t%s", vUsage)
    usg += fmt.Sprintf("\n")

    fmt.Println(usg)
}
```

This is pretty standard. You might want to change the default username/password. Ultimately we do not care about logging in, we just want to connect and get host info.

Init function

We setup flags, logging and check flags. `flag` package does not have `mutually_exclusive_group` from Python's `Argparse` package. It needs to be done manually. I will most likely move to a community cli package after this.

init function

```go
func init() {
    // Setup flags
    flag.StringVar(&out, "out", "", outUsage)
    flag.StringVar(&out, "o", "", outUsage)
    flag.StringVar(&in, "in", "", inUsage)
    flag.StringVar(&in, "i", "", inUsage)
    flag.Var(&targets, "targets", tUsage)
    flag.Var(&targets, "t", tUsage)
    flag.BoolVar(&verbose, "verbose", false, vUsage)
    flag.BoolVar(&verbose, "v", false, vUsage)

    // Set flag usage
    flag.Usage = usage

    // Parse flags
    flag.Parse()

    // Setting up logging
    logSSH = log.New(os.Stdout, LogPrefix, log.Ltime)

    // Check if we have enough arguments
    if len(os.Args) < 2 {
        flag.Usage()
        errorExit("not enough arguments", nil)
    }

    // Check if both in and targets are supported
    if (in != "") && (targets != nil) {
        errorExit("-in and -targets are mutually exclusive, use one", nil)
    }
}
```

`errorExit` just calls `logger.Fatalf` with a message. Logging the message and returning from main with status code 1.

errorExit

```go
// errorExit logs an error and then exits with status code 1.
func errorExit(m string, err error) {
    // If err is provided print it, otherwise don't
    if err != nil {
        logSSH.Fatalf("%v - stopping\n%v\n", m, err)
    }
    logSSH.Fatalf("%v - stopping\n", m)
}
```

Custom flag type

We are using a custom flag type for `-t`. This allows us to pass multiple addresses separated by `,` and get a slice of addresses directly. This is done through implementing the flag.value which contains two methods `String()` and `Set()`. In simple words:

1. Create a new type `mytype`.
2. Create two methods with `*mytype` receivers named `String()` and `Set()`.
   - `String()` casts the custom type to a `string` and returns it.
   - `Set(string)` has a `string` argument and populates the type, returns an error if applicable.
3. Create a new flag without an initial value:
   - Call `flag.NewFlagSet(&var,` instead of `flag.String(`.
   - Call `flag.Var(` instead of `flag.StringVar(` or `flag.IntVar(`.

I have written more about the `flag` package in [Hacking with Go - 03.1](#).

strList custom flag type

```go
// Custom flag type for -t (code re-used from flag section)
// Create a custom type from a string slice
type strList []string

// Implement String()
func (str *strList) String() string {
    return fmt.Sprintf("%v", *str)
}

// Implement Set(*strList)
func (str *strList) Set(s string) error {
    // If input was empty, return an error
    if s == "" {
        return errors.New("nil input")
    }
    // Split input by ","
    *str = strings.Split(s, ",")
    // Do not return an error
    return nil
}
```

SSHServer struct

We use a struct and some methods to hold server info. The `SSHServer` struct has these fields:

SSHServer struct

```go
// Struct to hold server data
type SSHServer struct {
    Address   string          // host:port
    Host      string          // IP address
    Port      int             // port
    IsSSH     bool            // true if server is running SSH on address:port
    Banner    string          // banner text, if any
    Cert      ssh.Certificate // server's certificate
    Hostname  string          // hostname
    PublicKey ssh.PublicKey   // server's public key
}
```

Not all fields will be populated. For example `Hostname` and `PublicKey` are only populated if the server responds with a public key. If it has a cert, then `Cert` will be populated instead.

New `*SSHServer`s are created by `NewSSHServer`.

NewSSHServer

```go
// NewSSHServer returns a new SSHServer with address, host and port populated.
// If address cannot be processed, an error will be returned.
func NewSSHServer(address string) (*SSHServer, error) {
    // Process address, return error if it's not in the correct format
    host, port, err := net.SplitHostPort(address)
    if err != nil {
        return nil, err
    }

    var s SSHServer

    s.Address = address
    s.Host = host
    s.Port, err = strconv.Atoi(port)
    if err != nil {
        return nil, err
    }
    // If port is not in (0,65535]
    if 0 > s.Port || s.Port > 65535 {
        return nil, errors.New(port + " invalid port")
    }
    return &s, nil
}
```

`net.SplitHostPort` splits `host:port` into two strings but it does not check the validity of either part. Meaning you can pass `500.500.500.500:70000` and it will be accepted because the format is correct.

To check if the IP is valid, we can use `net.ParseIP` and check the result (it's `nil` if it was not parsed correctly). However, we do not know if we are dealing with hostnames like `example.com:1234`. But we can check if ports are in the correct range.

SSHServers struct

`SSHServers` is a slice of `SSHServer` pointers. It has a Stringer method (a `String` method that returns a string representation of receiver).

SSHServers type and Stringer

```go
type SSHServers []*SSHServer

// String converts []*SSHServer to JSON. If it cannot convert to JSON, it
// will convert each member to string using fmt.Sprintf("%+v").
func (servers *SSHServers) String() string {
    var report string
    // Try converting to JSON
    report, err := ToJSON(servers, true)
    // If cannot convert to JSON
    if err != nil {
        // Save all servers as string (this is not as good as JSON)
        for _, v := range *servers {
            report += fmt.Sprintf("%+v\n%s\n", v, strings.Repeat("-", 30))
        }
        return report
    }
    return report
}
```

Struct to JSON

`ToJSON` converts a struct to a JSON string. If the second argument is `true`, it pretty prints it by indenting.

ToJSON

```go
// ToJSON converts input to JSON. If prettyPrint is set to True it will call
// MarshallIndent with 4 spaces.
// If your struct does not work here, make sure struct fields start with a
// capital letter. Otherwise they are not visible to the json package methods.
// We could also rewrite this as a method for ([]*SSHServer).
func ToJSON(s interface{}, prettyPrint bool) (string, error) {
    var js []byte
    var err error

    // Pretty print if specified
    if prettyPrint {
        js, err = json.MarshalIndent(s, "", "    ") // 4 spaces
    } else {
        js, err = json.Marshal(s)
    }

    // Check for marshalling errors
    if err != nil {
        return "", nil
    }

    return string(js), nil
}
```

This is one of the useful things I learned while working on this code. It's a pretty cool way of converting structs into strings. When printing with `"%+v"` format string, field pointers are not dereferenced and it will print the memory address. However, marshalling to JSON dereferences every field.

Note: When JSON-ing structs, make sure to mark fields as exportable by starting their names with capital letters. The JSON package cannot see them otherwise.

Utilities

There are a couple of misc functions.

`readTargetFile` reads addresses from a file (one address on each line) and returns a `[]string`.

`writeReport` gets a slice of `SSHServer` s (`SSHServers` to be exact), converts it to string (the Stringer we saw earlier will try to convert it to JSON first) and writes it to a file. The final file will be a JSON object that can be parsed.

Parsing SSH certificates <-- This is the important part

Inside ssh.ClientConfig there's a callback `HostKeyCallback`. This function should return `nil` if host is verified. Read Phil Pennock's blogpost Golang SSH Security for the history behind it.

Let's expand the tl;dr steps:

Step 1: Create ssh.CertChecker

We are interested in the following three ssh.CertChecker fields. All of them are callback functions:

CertChecker

```go
certCheck := &ssh.CertChecker{
    IsHostAuthority: hostAuthCallback(),
    IsRevoked:       certCallback(s),
    HostKeyFallback: hostCallback(s),
}
```

Don't worry about the functions for now. But remember these callback functions are only required to have a specific **return value but can have any number of arguments**. This is very useful we can pass our `SSHServer` objects and populate them inside these functions.

Step 2: Set Callback functions

Set callback functions for these three fields.

IsHostAuthority

`IsHostAuthority` must be defined. If not, we get a run-time error:

```
golang.org/x/crypto/ssh.(*CertChecker).CheckHostKey(0xc04206a140, 0xc0420080c0,
    0xc, 0x68d700, 0xc042058450, 0x68df80, 0xc0420a2000, 0x1, 0x8)
        Z:/Go/src/golang.org/x/crypto/ssh/certs.go:301 +0xae
golang.org/x/crypto/ssh.(*CertChecker).CheckHostKey-fm(0xc0420080c0, 0xc,
    0x68d700, 0xc042058450, 0x68df80, 0xc0420a2000, 0x0, 0x0)
        Z:/Go/src/hackingwithgo/04.5-01-ssh-harvester.go:205 +0x70
...
```

To discover the error cause, one must look at the source code for CheckHostKey. We'll see that `CheckHostKey` calls `IsHostAuthority`.

CertChecker.CheckHostKey source

```go
// CheckHostKey checks a host key certificate. This method can be
// plugged into ClientConfig.HostKeyCallback.
func (c *CertChecker) CheckHostKey(addr string, remote net.Addr, key PublicKey) err
    cert, ok := key.(*Certificate)
    if !ok {
        if c.HostKeyFallback != nil {
            return c.HostKeyFallback(addr, remote, key)
        }
        return errors.New("ssh: non-certificate host key")
    }
    if cert.CertType != HostCert {
        return fmt.Errorf("ssh: certificate presented as a host key has type %d", c
    }
    // If IsHostAuthority is not defined, run-time error occurs here
    if !c.IsHostAuthority(cert.SignatureKey, addr) {
        return fmt.Errorf("ssh: no authorities for hostname: %v", addr)
    }

    hostname, _, err := net.SplitHostPort(addr)
    if err != nil {
        return err
    }

    // Pass hostname only as principal for host certificates (consistent with OpenS
    return c.CheckCert(hostname, cert)
}
```

So what does this function do?

First it tries to get a certificate from `key PublicKey` (by casting). If the cast is not successful, it uses `HostKeyFallBack` to verity server's public key instead.

Then the function checks if the certificate type is `HostCert`. SSH differentiates between host and client certificates. For example OpenSSH's `keygen` uses the `-h` switch to sign and create a host key.

Another of our callbacks, `IsHostAuthority` is called next. If it returns `false`, the certificate is not valid. The docs say:

```
// IsHostAuthority should report whether the key is recognized as
// an authority for this host. This allows for certificates to be
// signed by other keys, and for those other keys to only be valid
// signers for particular hostnames. This must be set if this
// CertChecker will be checking host certificates.
```

This is just fancy talk for verifying the CA and performing certificate pinning. In other words we can check:

1. Is the certificate signed by *a valid CA*? Note, unlike TLS certs, most SSH certs are signed by internal CAs. Often we are relying on a hardcoded CA for verification.
2. Is the certificate signed by *the valid CA*? We don't want certs signed by other CAs.

`net.SplitHostPort` (we already used it above) splits `host:port` into `host` and `port` and passes `hostname` to `CheckCert`.

`CheckCert` does a couple of more checks. Most notably it calls another one of our functions `IsRevoked`.

CertChecker.CheckCert partial source

```go
// CheckCert checks CriticalOptions, ValidPrincipals, revocation, timestamp and
// the signature of the certificate.
func (c *CertChecker) CheckCert(principal string, cert *Certificate) error {
    if c.IsRevoked != nil && c.IsRevoked(cert) {
        return fmt.Errorf("ssh: certicate serial %d revoked", cert.Serial)
    }
    ...
```

IsHostAuthority callback

Not every function can be a callback function. Each function needs to return certain type. `IsHostAuthority` requires the callback function to have this return type:

- `func(ssh.PublicKey, string) bool`

In other words, our callback function needs to **return a function of that type**.

First we create a custom type (it's not defined in the package) and then create a function that returns that type:

hostAuthCallback

```go
// Define custom type for IsHostAuthority
type HostAuthorityCallBack func(ssh.PublicKey, string) bool

// hostAuthCallback is the callbackfunction for IsHostAuthority. Without
// it, ssh.CertChecker will not work.
func hostAuthCallback() HostAuthorityCallBack {
    // Return true because we just want to make this work
    return func(p ssh.PublicKey, addr string) bool {
        return true
    }
}
```

If we want the connection to continue, the internal function needs to return `true`.

IsRevoked

`IsRevoked` is not mandatory. If it's not set, it's ignored. Meaning there's no automatic certificate revocation checks happening without it. ~~Note the typo in the error message:~~ `certicate`. The typo has now been corrected. Honestly, I think this just means programs do not use this function (or I am terribly wrong and am using something which should not be used). If certificate is valid, this function must return `nil` or `false`.

IsRevoked callback

For the goal of grabbing the certificate and processing it, `IsRevoked` is the most useful. It gets the certificate as a parameter and we can do parse or verify it inside the function. `IsRevoked` must return:

- `func(cert *Certificate) bool`

Again we define that function type and declare our own function:

certCallback

```go
// Create IsRevoked function callback type
type IsRevokedCallback func(cert *ssh.Certificate) bool

// certCallback processes the SSH certificate. It is piggybacked on the
// IsRevoked callback function. It must return false (or nil) to keep the
// connection alive.
func certCallback(s *SSHServer) IsRevokedCallback {

    return func(cert *ssh.Certificate) bool {
        // Grab the certificate
        s.Cert = *cert
        s.IsSSH = true

        // Always return false
        return false
    }
}
```

Inside `IsRevoked` we have access to the SSH certificate. Here we just assign it to the `Cert` field.

If you want to verify the certificate, this is the place.

~~Question!!!!~~ Solved

~~Help me if you can. I don't like returning unnamed functions like this. But unless I create global variables, I need to be able to access~~ `s *SSHServer` ~~inside~~ `certCallback` ~~to populate it. The function type is strict so I cannot add arguments.~~

~~I think defining the inside function as a method will work. Am I write? Wrong? Please let me know if you know the answer.~~

Method is the way to go or just use anonymous functions. I don't like them but there's nothing wrong with using one.

HostKeyFallback

Not all servers have SSH certificates. In fact, most servers probably do not. If server does not send a certificate, this function will be called (and the connection will terminate if this function is not defined).

If server is valid this function should return nil.

hostCallback

```go
// hostCallback is the callback function for HostKeyCallback in SSH config.
// It can access hostname, remote address and server's public key.
func hostCallback(s *SSHServer) ssh.HostKeyCallback {
    return func(hostname string, remote net.Addr, key ssh.PublicKey) error {
        s.Hostname = hostname
        s.PublicKey = key
        // Return nil because we want the connection to move forward
        return nil
    }
}
```

Here we grab server's public key and hostname.

With these three callbacks set, we can move to the next step.

Step 3: Create ssh.ClientConfig

ssh.ClientConfig is needed for every SSH connection in Go. You can read about creating SSH connections in Hacking with Go - 04.4.

Sample ssh.ClientConfig

```go
// Create SSH config
config := &ssh.ClientConfig{
    // Test username and password
    User: TestUser,
    Auth: []ssh.AuthMethod{
        ssh.Password(TestPassword),
    },
    HostKeyCallback: certCheck.CheckHostKey,
    BannerCallback:  bannerCallback(s),
    Timeout:         Timeout, // timeout
}
```

`Timeout` is also important. we do not want goroutines to wait forever connecting to inaccessible addresses. It's set to 5 seconds by default. Can be changed in the constants.

Banner callback

Banner callback is another important function for information gathering. By now, you know the drill.

bannerCallback

```go
// bannerCallback is the callback function for BannerCallback in SSH config.
// Grabs server banner and stores it in the SSHServer object.
func bannerCallback(s *SSHServer) ssh.BannerCallback {
    return func(message string) error {
        // Store the banner
        s.Banner = message
        // Return nil because we want the connection to move forward
```

```
            return nil
        }
}
```

We store the banner message and return `nil`. Any other return value will terminate the connection.

Step 4: ClientConfig.HostKeyCallback

This callback starts the server verification chain. It needs a function with [ssh.HostKeyCallback]() type:

- `type HostKeyCallback func(hostname string, remote net.Addr, key PublicKey) error`

The package actually suggests [(*CertChecker) CheckHostKey]() (we looked at its source code earlier). Looking inside `ClientConfig`, you can see I am passing it like this:

- `HostKeyCallback: certCheck.CheckHostKey,`

This is where everything clicks. We created a `certCheck` and set its callback functions. Now we are passing it to be called when we connect to a server.

Other ways of verifying servers

If you do not want to verify server's certificate, you can plug in three different types of functions here.

- `ssh.FixedHostKey(key PublicKey)`: Returns a function to check the hostkey.
- `ssh.InsecureIgnoreHostKey()`: Ignore everything! **Danger! Will Robinson!**
- `Custom host verifier`: Return nil if host is ok, otherwise return an error.

Read more about them in the verifying host.

A note about ssh.InsecureIgnoreHostKey()
After the breaking change as a consequence of the Golang SSH security blog post linked earlier, everyone seems to be using this. I am not in the position to tell you how to write your code. But make sure you know what you are doing when using this function. *cough* hashicorp packer *cough*.

Step 5: Connecting to SSH servers

Here comes the concurrent part. We have a list of addresses and our callbacks are set correctly. Time to connect to servers with `discover`.

discover method

(s *SSHServer) discover

```go
// discover connects to ip:port and attempts to make an SSH connection.
// If successful, some SSH properties will be populated (most importantly isSSH
// and isAlive).
func (s *SSHServer) discover() {
    // Release waitgroup after returning
    defer discoveryWG.Done()

    defer logSSH.Println("finished connecting to", s.Address)

    certCheck := &ssh.CertChecker{
        IsHostAuthority: hostAuthCallback(),
        IsRevoked:       certCallback(s),
        HostKeyFallback: hostCallback(s),
    }

    // Create SSH config
    config := &ssh.ClientConfig{
        // Test username and password
        User: TestUser,
        Auth: []ssh.AuthMethod{
            ssh.Password(TestPassword),
        },
        HostKeyCallback: certCheck.CheckHostKey,
        BannerCallback:  bannerCallback(s),
        Timeout:         Timeout, // timeout
    }

    logSSH.Println("starting SSH connection to ", s.Address)
    sshConn, err := ssh.Dial("tcp", s.Address, config)
    if err != nil {
        // If error contains "unable to authenticate", there's something there
        logSSH.Println("error ", err)
        return
    }

    // Close connection if we succeed (almost never happens)
    sshConn.Close()
}
```

First we defer releasing the waitgroup and the log message. This waitgroup will be explained later. In short, it's here to ensure that all `discover` goroutines are finished before starting the next stage.

Next are `CertCheck` and `ClientConfig`. We have already seen them. And finally we are connecting with `ssh.Dial`.

Goroutines and sync.WaitGroups

Each connection is done in its own goroutine. This means, we have to wait for these to complete before processing the results. We use `sync.WaitGroups`. For a longer version please read [Hacking with Go - 02.6 - Syncing goroutines](). But a tl;dr description is:

1. Every time a goroutine is started, we add one to the waitgroup (note we need to do this in the calling function, not inside the goroutine).
2. When the goroutine returns we subtract one (the `defer discoveryWG.Done()` in `discover`).
3. Wait in main for all goroutines to finish with `discoveryWG.Wait()`. This will block the program until they all return.

Syncing goroutines

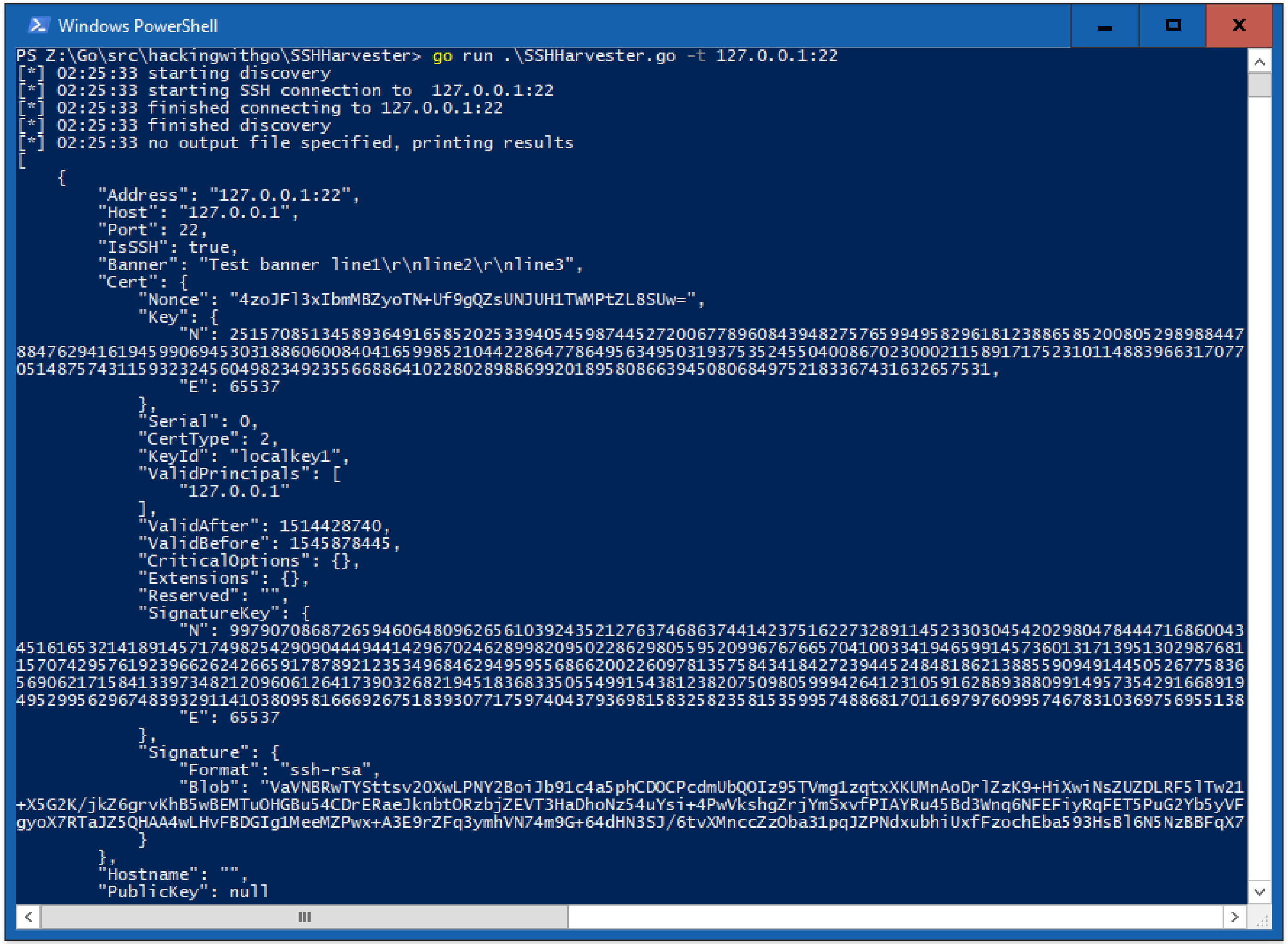
```go
for _, v := range servers {
    // Before each goroutine add 1 to waitgroup
    discoveryWG.Add(1)
    go v.discover()
}

// Wait for all discovery goroutines to finish
discoveryWG.Wait()
```

SSH Harvester in action

And finally we can see the tool in action.

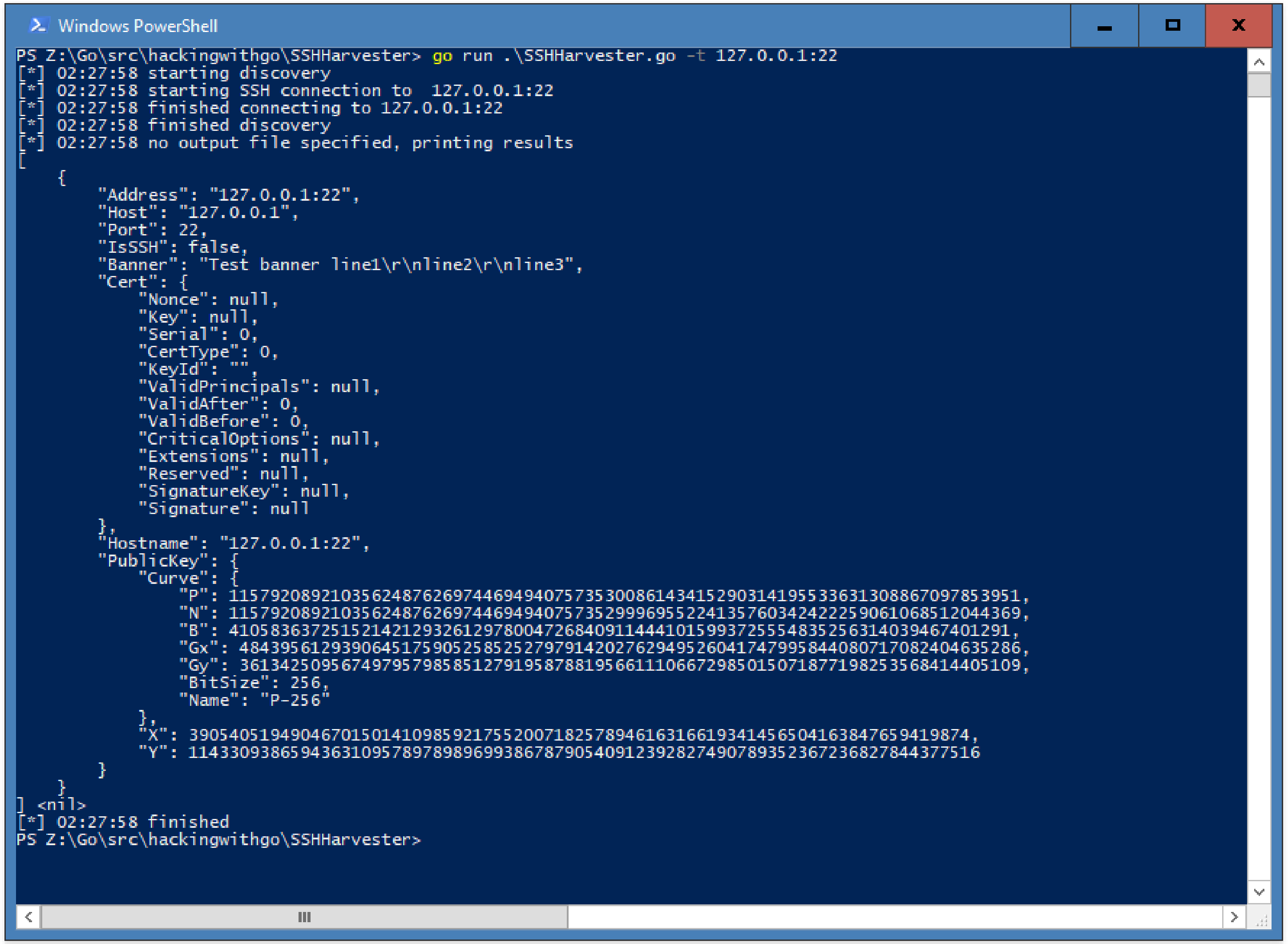
If the server returns a certificate:



```
PS Z:\Go\src\hackingwithgo\SSHHarvester> go run .\SSHHarvester.go -t 127.0.0.1:22
[*] 02:25:33 starting discovery
[*] 02:25:33 starting SSH connection to  127.0.0.1:22
[*] 02:25:33 finished connecting to 127.0.0.1:22
[*] 02:25:33 finished discovery
[*] 02:25:33 no output file specified, printing results
[
    {
        "Address": "127.0.0.1:22",
        "Host": "127.0.0.1",
        "Port": 22,
        "IsSSH": true,
        "Banner": "Test banner line1\r\nline2\r\nline3",
        "Cert": {
            "Nonce": "4zoJFl3xIbmMBZyoTN+Uf9gQZsUNJUH1TWMPtZL8SUw=",
            "Key": {
                "N": 251570851345893649165852025339405459874452720067789608439482757659949582961812388658520080529898844 7
884762941619459906945303188606008404165998521044228647786495634950319375352455040086702300021158917175231011488396631707 7
051487574311593232456049823492355668864102280289886992018958086639450806849752183367431632657531,
                "E": 65537
            },
            "Serial": 0,
            "CertType": 2,
            "KeyId": "localkey1",
            "ValidPrincipals": [
                "127.0.0.1"
            ],
            "ValidAfter": 1514428740,
            "ValidBefore": 1545878445,
            "CriticalOptions": {},
            "Extensions": {},
            "Reserved": "",
            "SignatureKey": {
                "N": 997907086872659460648096265610392435212763746863744142375162273289114523303045420298047844471686004 3
451616532141891457174982542909044494414296702462899820950228629805595209967676657041003341946599145736013171395130298768 1
157074295761923966262426659178789212353496846294959556866200226097813575843418427239445248481862138855909491445052677583 6
569062171584133973482120960612641739032682194518368335055499154381238207509805999426412310591628893880991495735429166891 9
495299562967483932911410380958166692675183930771759740437936981583258235815359957488681701169797609957467831036975695513 8
                "E": 65537
            },
            "Signature": {
                "Format": "ssh-rsa",
                "Blob": "VaVNBRwTYSttsv2OXwLPNY2BoiJb91c4a5phCDOCPcdmUbQOIz95TVmg1zqtxXKUMnAoDrlZzK9+HiXwiNsZUZDLRF5lTw21
+X5G2K/jkZ6grvKhB5wBEMTuOHGBu54CDrERaeJknbtORzbjZEVT3HaDhoNz54uYsi+4PwVkshgZrjYmSxvfPIAYRu45Bd3Wnq6NFEFiyRqFET5PuG2Yb5yVF
gyoX7RTaJZ5QHAA4wLHvFBDGIg1MeeMZPwx+A3E9rZFq3ymhVN74m9G+64dHN3SJ/6tvXMnccZzOba31pqJZPNdxubhiUxfFzochEba593HsBl6N5NzBBFqX7
            }
        },
        "Hostname": "",
        "PublicKey": null
```

SSH certificate info

If it returns a public key, `HostKeyFallBack` is triggered and we can it:



```
PS Z:\Go\src\hackingwithgo\SSHHarvester> go run .\SSHHarvester.go -t 127.0.0.1:22
[*] 02:27:58 starting discovery
[*] 02:27:58 starting SSH connection to  127.0.0.1:22
[*] 02:27:58 finished connecting to 127.0.0.1:22
[*] 02:27:58 finished discovery
[*] 02:27:58 no output file specified, printing results
[
    {
        "Address": "127.0.0.1:22",
        "Host": "127.0.0.1",
        "Port": 22,
        "IsSSH": false,
        "Banner": "Test banner line1\r\nline2\r\nline3",
        "Cert": {
            "Nonce": null,
            "Key": null,
            "Serial": 0,
            "CertType": 0,
            "KeyId": "",
            "ValidPrincipals": null,
            "ValidAfter": 0,
            "ValidBefore": 0,
            "CriticalOptions": null,
            "Extensions": null,
            "Reserved": null,
            "SignatureKey": null,
            "Signature": null
        },
        "Hostname": "127.0.0.1:22",
        "PublicKey": {
            "Curve": {
                "P": 115792089210356248762697446949407573530086143415290314195533631308867097853951,
                "N": 115792089210356248762697446949407573529996955224135760342422259061068512044369,
                "B": 41058363725152142129326129780047268409114441015993725554835256314039467401291,
                "Gx": 48439561293906451759052585252797914202762949526041747995844080717082404635286,
                "Gy": 36134250956749795798585127919587881956611106672985015071877198253568414405109,
                "BitSize": 256,
                "Name": "P-256"
            },
            "X": 39054051949046701501410985921755200718257894616316619341456504163847659419874,
            "Y": 114330938659436310957897898969938678790540912392827490789352367236827844377516
        }
    }
] <nil>
[*] 02:27:58 finished
PS Z:\Go\src\hackingwithgo\SSHHarvester>
```

SSH public key

Note, server's have different keys for different ciphersuits. For example `dsa`, `ecdsa`, `rsa` and `ed25519` (the DJB curve). Depending what ciphersuite client supports, you may see one of these. That's another TODO.

Conclusion

It took me a couple of days to figure everything out because I could not find any examples or tutorials. But now we know how to verify SSH certificates. Hope this is useful, if you have any feedback please let me know.

---

1. I should have actually sent a patch. But signing up for Gerrit was a pain. Would have been the easiest way to become a "Golang contributor" and put it in my Twitter bio/resume (kidding). [return]

Posted by Parsia • Dec 29, 2017 • Tags: Golang SSH

Windows XP 32-bit SP3 Virtual Machines
Decoding Large Base64 Files with Go

3 Comments Parsiya 1 Login

Recommend 1 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Dave Mazzoni • 6 months ago

Parsia,
Thank you for leaving such an extensive post on ssh coding. I wish all gophers could leave such beautiful detail in a post. Great job!

Reply • Share ›

Dmitry Verkhoturov • 2 years ago

Quite interesting. Login\password part is confusing, I guess it should be clarified what you don't need them at all for this study case.

Reply • Share ›

Parsia Parsia ↪ Dmitry Verkhoturov • 2 years ago

Good point, I need to clarify.

Reply • Share ›

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS