badtouch Documentation

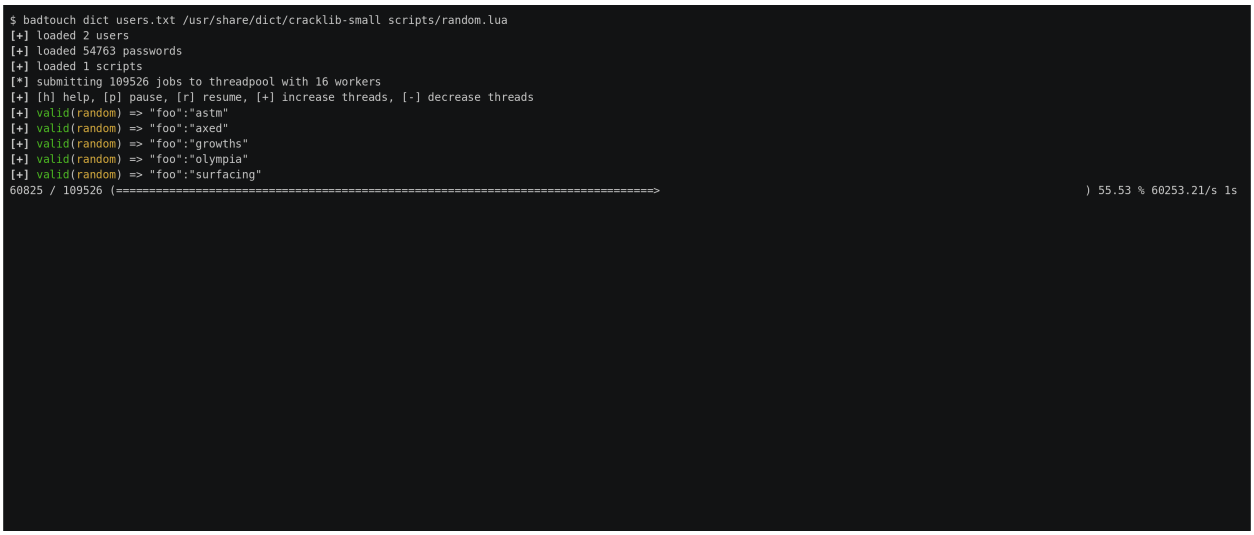
kpcyrd

Jan 14, 2021

Contents

badtouch is a scriptable network authentication cracker. While the space for common service bruteforce is already very well saturated, you may still end up writing your own python scripts when testing credentials for web applications.

The scope of badtouch is specifically cracking custom services. This is done by writing scripts that are loaded into a lua runtime. Those scripts represent a single service and provide a `verify(user, password)` function that returns either true or false. Concurrency, progress indication and reporting is magically provided by the badtouch runtime.



CHAPTER 1

Getting Started

1.1 Installation

If available, please prefer the package shipped by your linux distribution.

1.1.1 Archlinux

```
$ pacman -S badtouch
```

1.1.2 Mac OSX

```
$ brew install badtouch
```

1.1.3 Docker

```
$ docker run --rm kpcyrd/badtouch --help
```

1.1.4 Source

To build from source, make sure you have rust and `libssl-dev` installed.

```
$ git clone https://github.com/kpcyrd/badtouch
$ cd badtouch
$ cargo build
```

1.2 Usage

1.2.1 Options

```
-n, --workers <workers>     The number of concurrent workers to run.
-o, --output <output>       Write results to this file.
-v, --verbose               Enable verbose output.
-h, --help                  Prints help information.
-V, --version               Prints version information.
```

1.2.2 Dictionary attack

Try each password for each user with every script.

```
badtouch dict <users> <passwords> [scripts]...
```

1.2.3 Credential confirmation

Load a list of credentials with the format `user:password` and verify them with every script.

```
badtouch creds <credentials> [scripts]...
```

1.2.4 Username enumeration

Takes a list of username and verifies they exist on the system. This is still executing the `verify` function with two arguments, but the password is set to `nil`. You may write a script that can do both by checking the password for nil to detect in which mode the script is executed.

```
badtouch enum <users> [scripts]...
```

1.2.5 Oneshot

Test a single username-password combination using a specific script. This command is also useful when developing a new script. If the password argument is omitted, the script is executed in enumerate mode. If you want to use this command in scripts, set `-x` so the exitcode is set to 2 if the credentials are invalid.

```
badtouch oneshot [-x] <script> <user> [password]
```

1.3 Scripting

A simple script could look like this:

```
descr = "example.com"

function verify(user, password)
    session = http_mksession()
```

(continues on next page)

(continued from previous page)

```
    -- get csrf token
    req = http_request(session, 'GET', 'https://example.com/login', {})
    resp = http_send(req)
    if last_err() then return end

    -- parse token from html
    html = resp['text']
    csrf = html_select(html, 'input[name="csrf"]')
    token = csrf["attrs"]["value"]

    -- send login
    req = http_request(session, 'POST', 'https://example.com/login', {
        form={
            user=user,
            password=password,
            csrf=token
        }
    })
    resp = http_send(req)
    if last_err() then return end

    -- search response for successful login
    html = resp['text']
    return html:find('Login successful') ~= nil
end
```

Please see the reference and [examples](/scripts) for all available functions. Keep in mind that you can use *print(x)* and *badtouch oneshot* to debug your script.

1.3.1 base64_decode

Decode a base64 string.

```
base64_decode("ww==")
```

1.3.2 base64_encode

Encode a binary array with base64.

```
base64_encode("\x00\xff")
```

1.3.3 clear_err

Clear all recorded errors to prevent a requeue.

```
if last_err() then
    clear_err()
    return false
else
    return true
end
```

1.3.4 execve

Execute an external program. Returns the exit code.

```
execve("myprog", {"arg1", "arg2", "--arg", "3"})
```

1.3.5 hex

Hex encode a list of bytes.

```
hex("\x6F\x68\x61\x69\x0A\x00")
```

1.3.6 hmac_md5

Calculate an hmac with md5. Returns a binary array.

```
hmac_md5("secret", "my authenticated message")
```

1.3.7 hmac_sha1

Calculate an hmac with sha1. Returns a binary array.

```
hmac_sha1("secret", "my authenticated message")
```

1.3.8 hmac_sha2_256

Calculate an hmac with sha2_256. Returns a binary array.

```
hmac_sha2_256("secret", "my authenticated message")
```

1.3.9 hmac_sha2_512

Calculate an hmac with sha2_512. Returns a binary array.

```
hmac_sha2_512("secret", "my authenticated message")
```

1.3.10 hmac_sha3_256

Calculate an hmac with sha3_256. Returns a binary array.

```
hmac_sha3_256("secret", "my authenticated message")
```

1.3.11 hmac_sha3_512

Calculate an hmac with sha3_512. Returns a binary array.

```
hmac_sha3_512("secret", "my authenticated message")
```

1.3.12 html_select

Parses an html document and returns the first element that matches the css selector. The return value is a table with `text` being the inner text and `attrs` being a table of the elements attributes.

```
csrf = html_select(html, 'input[name="csrf"]')
token = csrf["attrs"]["value"]
```

1.3.13 html_select_list

Same as *html_select* but returns all matches instead of the first one.

```
html_select_list(html, 'input[name="csrf"]')
```

1.3.14 http_basic_auth

Sends a `GET` request with basic auth. Returns `true` if no `WWW-Authenticate` header is set and the status code is not `401`.

```
http_basic_auth("https://httpbin.org/basic-auth/foo/buzz", user, password)
```

1.3.15 http_mksession

Create a session object. This is similar to `requests.Session` in python-requests and keeps track of cookies.

```
session = http_mksession()
```

1.3.16 http_request

Prepares an http request. The first argument is the session reference and cookies from that session are copied into the request. After the request has been sent, the cookies from the response are copied back into the session.

The next arguments are the `method`, the `url` and additional options. Please note that you still need to specify an empty table `{}` even if no options are set. The following options are available:

- `query` - a map of query parameters that should be set on the url
- `headers` - a map of headers that should be set
- `basic_auth` - configure the basic auth header with `{"user, "password"}`
- `user_agent` - overwrite the default user agent with a string
- `json` - the request body that should be json encoded
- `form` - the request body that should be form encoded

- `body` - the raw request body as string

```
req = http_request(session, 'POST', 'https://httpbin.org/post', {
    json={
        user=user,
        password=password,
    }
})
resp = http_send(req)
if last_err() then return end
if resp["status"] ~= 200 then return "invalid status code" end
```

1.3.17 http_send

Send the request that has been built with *http_request*. Returns a table with the following keys:

- `status` - the http status code
- `headers` - a table of headers
- `text` - the response body as string

```
req = http_request(session, 'POST', 'https://httpbin.org/post', {
    json={
        user=user,
        password=password,
    }
})
resp = http_send(req)
if last_err() then return end
if resp["status"] ~= 200 then return "invalid status code" end
```

1.3.18 json_decode

Decode a lua value from a json string.

```
json_decode("{\"data\":{\"password\":\"fizz\",\"user\":\"bar\"},\"list\":[1,3,3,7]}")
```

1.3.19 json_encode

Encode a lua value to a json string. Note that empty tables are encoded to an empty object `{}` instead of an empty list `[]`.

```
x = json_encode({
    hello="world",
    almost_one=0.9999,
    list={1,3,3,7},
    data={
        user=user,
        password=password,
        empty=nil
    }
})
```

1.3.20 last_err

Returns `nil` if no error has been recorded, returns a string otherwise.

```
if last_err() then return end
```

1.3.21 ldap_bind

Connect to an ldap server and try to authenticate with the given user.

```
ldap_bind("ldaps://ldap.example.com/",
    "cn=\"" .. ldap_escape(user) .. "\",ou=users,dc=example,dc=com", password)
```

1.3.22 ldap_escape

Escape an attribute value in a relative distinguished name.

```
ldap_escape(user)
```

1.3.23 ldap_search_bind

Connect to an ldap server, log into a search user, search for the target user and then try to authenticate with the first DN that was returned by the search.

```
ldap_search_bind("ldaps://ldap.example.com/",
    -- the user we use to find the correct DN
    "cn=search_user,ou=users,dc=example,dc=com", "searchpw",
    -- base DN we search in
    "dc=example,dc=com",
    -- the user we test
    user, password)
```

1.3.24 md5

Hash a byte array with md5 and return the results as bytes.

```
hex(md5("\x00\xff"))
```

1.3.25 mysql_connect

Connect to a mysql database and try to authenticate with the provided credentials. Returns a mysql connection on success.

```
sock = mysql_connect("127.0.0.1", 3306, user, password)
```

1.3.26 mysql_query

Run a query on a mysql connection. The 3rd parameter is for prepared statements.

```
rows = mysql_query(sock, 'SELECT VERSION(), :foo as foo', {
    foo='magic'
})
```

1.3.27 print

Prints the value of a variable. Please note that this bypasses the regular writer and may interfer with the progress bar. Only use this for debugging.

```
print({
    data={
        user=user,
        password=password
    }
})
```

1.3.28 rand

Returns a random `u32` with a minimum and maximum constraint. The return value can be greater or equal to the minimum boundary, and always lower than the maximum boundary. This function has not been reviewed for cryptographic security.

```
rand(0, 256)
```

1.3.29 randombytes

Generate the specified number of random bytes.

```
randombytes(16)
```

1.3.30 sha1

Hash a byte array with sha1 and return the results as bytes.

```
hex(sha1("\x00\xff"))
```

1.3.31 sha2_256

Hash a byte array with sha2_256 and return the results as bytes.

```
hex(sha2_256("\x00\xff"))
```

1.3.32 sha2_512

Hash a byte array with sha2_512 and return the results as bytes.

```
hex(sha2_512("\x00\xff"))
```

1.3.33 sha3_256

Hash a byte array with sha3_256 and return the results as bytes.

```
hex(sha3_256("\x00\xff"))
```

1.3.34 sha3_512

Hash a byte array with sha3_512 and return the results as bytes.

```
hex(sha3_512("\x00\xff"))
```

1.3.35 sleep

Pauses the thread for the specified number of seconds. This is mostly used to debug concurrency.

```
sleep(3)
```

1.3.36 sock_connect

Create a tcp connection.

```
sock = sock_connect("127.0.0.1", 1337)
```

1.3.37 sock_send

Send data to the socket.

```
sock_send(sock, "hello world")
```

1.3.38 sock_recv

Receive up to 4096 bytes from the socket.

```
x = sock_recv(sock)
```

1.3.39 sock_sendline

Send a string to the socket. A newline is automatically appended to the string.

```
sock_sendline(sock, line)
```

1.3.40 sock_recvline

Receive a line from the socket. The line includes the newline.

```
x = sock_recvline(sock)
```

1.3.41 sock_recvall

Receive all data from the socket until EOF.

```
x = sock_recvall(sock)
```

1.3.42 sock_recvline_contains

Receive lines from the server until a line contains the needle, then return this line.

```
x = sock_recvline_contains(sock, needle)
```

1.3.43 sock_recvline_regex

Receive lines from the server until a line matches the regex, then return this line.

```
x = sock_recvline_regex(sock, "^250 ")
```

1.3.44 sock_recvn

Receive exactly n bytes from the socket.

```
x = sock_recvn(sock, 4)
```

1.3.45 sock_recvuntil

Receive until the needle is found, then return all data including the needle.

```
x = sock_recvuntil(sock, needle)
```

1.3.46 sock_sendafter

Receive until the needle is found, then write data to the socket.

```
sock_sendafter(sock, needle, data)
```

1.3.47 sock_newline

Overwrite the default *n* newline.

```
sock_newline(sock, "\r\n")
```

1.3.48 Wrapping python scripts

The badtouch runtime is still very bare bones, so you might have to shell out to your regular python script occasionally. Your wrapper may look like this:

```
descr = "example.com"

function verify(user, password)
    ret = execve("./docs/test.py", {user, password})
    if last_err() then return end

    if ret == 2 then
        return "script signaled an exception"
    end

    return ret == 0
end
```

Your python script may look like this:

```
import sys

try:
    if sys.argv[1] == "foo" and sys.argv[2] == "bar":
        # correct credentials
        sys.exit(0)
    else:
        # incorrect credentials
        sys.exit(1)
except:
    # signal an exception
    # this requeues the attempt instead of discarding it
    sys.exit(2)
```

1.4 Configuration

You can place a config file at `~/.config/badtouch.toml` to set some defaults.

1.4.1 Global user agent

```
[runtime]
user_agent = "w3m/0.5.3+git20180125"
```

1.4.2 RLIMIT_NOFILE

```
[runtime]
# requires CAP_SYS_RESOURCE
# sudo setcap 'CAP_SYS_RESOURCE=+ep' /usr/bin/badtouch
rlimit_nofile = 64000
```