Chapter 4: Web Reconnaissance in Rust

Introduction

The year 2010 marked a pivotal moment in cybersecurity, witnessing the emergence of sophisticated cyber threats through events like Operation Aurora and the Stuxnet attack. Operation Aurora targeted multinational businesses and Gmail accounts, while Stuxnet, a technologically advanced assault, focused on SCADA systems in Iran. Both incidents underscored the role of social engineering in facilitating infiltration, as highlighted by Constantin in 2012. Since then, the landscape of cybersecurity has continued to evolve, with notable recent attacks adding to the complexity of the threat landscape. Examples include the SolarWinds supply chain attack in 2020, where malicious actors compromised software updates to distribute malware, and the Colonial Pipeline ransomware attack in 2021, which disrupted fuel supply in the United States. These incidents underscore the persistent relevance of social engineering and the ongoing need for robust cybersecurity measures in the face of ever-evolving cyber threats.

As we navigate the complexities of web reconnaissance, the enduring significance of the human element in cybersecurity becomes apparent. Regardless of the technological sophistication of cyber attacks, effective social engineering remains a critical force multiplier, amplifying the impact of these threats. In the following sections, we'll delve into how Rust, with its emphasis on memory safety and performance, can be leveraged to automate web reconnaissance and enhance social engineering attacks.

1. Web Exploration in Rust

In the world of web technologies, the ability to browse the Internet anonymously stands as a fundamental skill. Rust's `Reqwest` library, a powerful crate in the Rust ecosystem, provides the means to achieve this objective. In our exploration, we will delve into the details of utilizing `reqwest` for (anonymous) web browsing, underscoring the importance of comprehending the mechanisms governing online anonymity.

Rust Reqwest offers powerful capabilities for web interaction with a focus on performance and safety allowing for seamless manipulation of browser elements. An example script below showcases basic usage, retrieving and printing the HTML source code of a specified website:

```rust
use reqwest;

async fn fetch_page(url: &str) -> Result<(), reqwest::Error> {
    let body = reqwest::get(url)
        .await?
        .text()
        .await?;
    println!("{}", body);
    Ok(())
}

fn main() {
    fetch_page("http://www.google.com").await.unwrap();
}
```

In this script, Reqwest's `get` method retrieves the webpage, and the `text` method extracts the HTML source code. This foundational knowledge forms the basis for more advanced web reconnaissance techniques.

```
:dep reqwest = { version="0.11.23", features=["cookies",] }
```

```
use reqwest;

async fn fetch_page(url: &str) -> Result<(), reqwest::Error> {
    let body = reqwest::get(url)
        .await?
        .text()
        .await?;
    println!("{}", body);
    Ok(())
}

fetch_page("http://www.google.com").await.unwrap();
```

```
<!doctype html><html dir="rtl" itemscope="" itemtype="http://schema.org/WebPage" lang="ar-LE
var h=this||self;function l(){return void 0!==window.google&&void 0!==window.google.kOPI&&0!
function t(a,b,c,d,k){var e="";-1===b.search("&ei=")&&(e="&ei="+p(d),-1===b.search("&lei=")&
document.documentElement.addEventListener("submit",function(b){var a;if(a=b.target){var c=a.
.gbqfb,.gbqfba,.gbqfbb{-moz-border-radius:2px;-webkit-border-radius:2px;border-radius:2px;cu
#gbmpas{max-height:220px}#gbmm{max-height:530px}.gbsb{-webkit-box-sizing:border-box;display:
</style><style>body,td,a,p,.h{font-family:arial,sans-serif}body{margin:0;overflow-y:scroll}#
var h=this||self;var k,l=null!=(k=h.mei)?k:1,n,p=null!=(n=h.sdo)?n:!0,q=0,r,t=google.erd,v=t
b(t.bv);var f=a.lineNumber;void 0!==f&&(c+="&line="+f);var g=a.fileName;g&&(0<g.indexOf("-ex

 Copyright The Closure Library Authors.
 SPDX-License-Identifier: Apache-2.0
*/
var e=this||self;var aa=function(a,b,c,d){d=d||{};d._sn=["cfg",b,c].join(".");window.gbar.lo
function ea(a,b,c){var d="on"+b;if(a.addEventListener)a.addEventListener(b,c,!1);else if(a.a
var ma={},y=[],na=h.b("0.1",.1),oa=h.a("1",!0),pa=function(a,b){y.push([a,b])},qa=function(a
document.getElementsByTagName("body")[0]||document.getElementsByTagName("head")[0]).appendCh
n("mdd","");n("has",ra);n("trh",ta);n("tev",D);if(h.a("m;/_/scs/abc-static/_/js/k=gapi.gapi.
d.i()}f.dgl(a,b)},L=window.___jsl=H(window.___jsl,{});L.h=H(L.h,"m;/_/scs/abc-static/_/js/k=
function _mlToken(a,b){try{if(1>Ca){Ca++;var c=a;b=b||{};var d=encodeURIComponent,g=["//www.
```

```
"og."+b._sn);for(var k in b)g.push("&"),g.push(d(k)),g.push("="),g.push(d(b[k]));g.push("&en
[Ma?"":"https://www.gstatic.com","/og/_/js/d=1/k=","og.og.en_US.QyC5j4_7WQo.es5.O","/rt=j/m=
var Wa=function(){for(var a=[],b,c=0;b=Qa[c];++c)(b=document.getElementById(b))&&a.push(b);r
O(k,"gbto");else{if(S){var m=document.getElementById(S);if(m&&m.getAttribute){var p=m.getAtt
a.currentStyle.direction:a.style.direction;return"rtl"==b},hb=function(a,b,c){if(a)try{var c
k.childNodes[d]||null);g=!0;break}}if(g){if(d+1<k.childNodes.length){var Fa=k.childNodes[d+1
b){for(var c=b.length,d=0;d<c;d++)if(M(a,b[d]))return!0;return!1},ib=function(a,b,c){hb(a,b,
a.preventDefault();a.returnValue=!1;a.cancelBubble=!0},rb=null,bb=function(a,b){T();if(a){sb
b[g];g++){var k=document.createElement("div");k.innerHTML=c;d.appendChild(k)}}else d.innerHT
n("so",Xa);n("sos",Wa);n("si",Ya);n("tg",db);n("close",eb);n("rdd",fb);n("addLink",ib);n("ad
Tb=function(){B(function(){f.spd()})};n("spn",Ob);n("spp",Qb);n("sps",Pb);n("spd",Tb);n("paa
if(h.a("")){var Ub="prf",Vb={d:h.a(""),e:"",sanw:h.a(""),p:"https://lh3.googleusercontent.co
ppm:"&#1589;&#1601;&#1581;&#1577; Google+"};w[Ub]=Vb};var V,Wb,W,Xb,X=0,Yb=function(a,b,c){i
function(a,b,c){if(Y([1],"aop")&&c){if(W)for(var d in W)W[d]=W[d]&&-1!=Yb(c,d);else for(W={}
var dc=function(a){var b=!1;try{b=a.cookie&&a.cookie.match("PREF")}catch(c){}return!b},ec=fu
c||(b="og-up-"+b);if(ec())return e.localStorage.getItem(b);if(fc(a))return a.load(a.id),a.ge
b+"-([0-9]+):"));if(d&&d[1])return parseInt(d[1],10)}}catch(g){g.code!=DOMException.QUOTA_EX
(function(){try{var b=window.gbar.i.i;var c=window.gbar;var f=function(d){try{var a=document
(function(){try{var a=window.gbar;a.mcf("pm",{p:""});}catch(e){window.gbar&&gbar.logger&&gba
(function(){try{var a=window.gbar;a.mcf("mm",{s:"1"});}catch(e){window.gbar&&gbar.logger&&gb
(function(){try{var d=window.gbar.i.i;var e=window.gbar;var f=e.i;var g=f.c("1",0),h=/\bgbmt
n(a)},q=function(){if(window.google&&window.google.sn){var a=/.*hp$/;return a.test(window.go
(function(){try{/*

 Copyright The Closure Library Authors.
 SPDX-License-Identifier: Apache-2.0
*/
var a=this||self;var b=window.gbar;var c=b.i;var d=c.a,e=c.c,f={cty:"LBN",cv:"591754533",dbg
snid:e("28834"),to:e("300000"),u:e(""),vf:".66."},g="bndcfg",h=f,k=g.split("."),l=a;k[0]in l
(function(){try{window.gbar.rdl();}catch(e){window.gbar&&gbar.logger&&gbar.logger.ml(e,{"_sn
</script></head><body bgcolor="#fff"><script nonce="M2mhXcT2weLVc33PCU9lFg">(function(){var
if (!iesg){document.f&&document.f.q.focus();document.gbqf&&document.gbqf.q.focus();}
}
})();</script><div id="mngb"><div id=gb dir=rtl class="gbrtl"><script nonce='M2mhXcT2weLVc33
```

Web reconnaissance extends beyond mere webpage retrieval. With `reqwest`, we can delve into advanced techniques such as handling cookies, managing user-agents, and employing proxies for enhanced anonymity. Let's explore how these features can be leveraged to craft a more sophisticated and stealthy web reconnaissance tool in Rust.

1.1.1 Managing Cookies for Persistent Sessions Cookies play a crucial role in web sessions, and their strategic management is essential for maintaining persistent connections during reconnaissance. In Rust, Reqwest provides robust support for handling cookies. The following example demonstrates how to utilize Reqwest's cookie jar for managing and maintaining cookies across multiple requests:

```rust
use reqwest;

async fn browse_with_cookies(url: &str) -> Result<(), reqwest::Error> {
    let client = reqwest::Client::builder()
        .cookie_store(true)
        .build()?;

    let response = client.get(url).send().await?;
    let cookies = response.cookies();

    let new_response = client.get("http://www.google.com/another-page").send().await?;

    println!("{}", new_response
        .text()
        .await?);

    Ok(())
}

browse_with_cookies("http://www.google.com").await.unwrap();
```

In this example, we create a `Reqwest` client with a cookie jar, make a request to a website, store the received cookies, and later use these cookies in a subsequent request. This allows for the maintenance of a persistent session across multiple interactions.

```rust
async fn browse_with_cookies(url: &str) -> Result<(), reqwest::Error> {
    let client = reqwest::Client::builder()
        .cookie_store(true)
        .build()?;

    let response = client.get(url).send().await?;
    let cookies = response.cookies();

    let new_response = client.get("http://www.google.com/another-page").send().await?;

    println!("{}", new_response
        .text()
        .await?);

    Ok(())
}


browse_with_cookies("http://www.google.com").await.unwrap();
```

```
<!DOCTYPE html>
<html lang=en>
  <meta charset=utf-8>
  <meta name=viewport content="initial-scale=1, minimum-scale=1, width=device-width">
  <title>Error 404 (Not Found)!!1</title>
  <style>
    *{margin:0;padding:0}html,code{font:15px/22px arial,sans-serif}html{background:#fff;colo
  </style>
  <a href=//www.google.com/><span id=logo aria-label=Google></span></a>
  <p><b>404.</b> <ins>That's an error.</ins>
  <p>The requested URL <code>/another-page</code> was not found on this server.  <ins>That's
```

1.1.2 Crafting Stealthy Requests with User-Agents User-agents convey information about the client's browser and system to the server. Crafting stealthy requests involves manipulating the user-agent string to mimic various browsers or devices. In Rust, Reqwest facilitates this through its header manipulation capabilities. The following example illustrates how to set a custom user-agent header in a Reqwest request:

```
use reqwest;

async fn browse_with_custom_user_agent(url: &str) -> Result<(), reqwest::Error> {
    let client = reqwest::Client::builder()
        .user_agent("Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, li
        .build()?;

    let response = client.get(url).send().await?;

    println!("{}", response.text().await?);

    Ok(())
}

fn main() {
    browse_with_custom_user_agent("http://www.google.com/invalid").await.unwrap();
}
```

In this example, we create a Reqwest client with a custom user-agent header, making the request appear as if it originates from a specific browser and platform. This manipulation enhances stealth and reduces the likelihood of detection.

```
+------------------------+
|                        |
|  +------------------+  |
|  |   Manipulate     |  |
|  |   User-Agent     |  |
|  +------------------+  |
|          |             |
|          V             |
|   +--------------+     |
|   |  Request     |     |
|   |  (Custom     |     |
|   |  User-Agent) |     |
|   +--------------+     |
|          |             |
|          V             |
|   +--------------+     |
|   |  HTTP        |     |
|   |  Request     |     |
|   |  Sent        |     |
|   +--------------+     |
|          |             |
|          V             |
|   +--------------+     |
|   |  Server      |     |
|   |  Response    |     |
|   +--------------+     |
|          |             |
|          V             |
|   +--------------+     |
|   |  Print       |     |
|   |  Response    |     |
|   +--------------+     |
|                        |
+------------------------+
```

```rust
use reqwest;

async fn browse_with_custom_user_agent(url: &str) -> Result<(), reqwest::Error> {
    let client = reqwest::Client::builder()
        .user_agent("Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, li
        .build()?;
    let response = client.get(url).send().await?;

    println!("{}", response.text().await?);

    Ok(())
}


browse_with_custom_user_agent("http://www.google.com/invalid").await.unwrap();
```

```
<!DOCTYPE html>
<html lang=en>
  <meta charset=utf-8>
  <meta name=viewport content="initial-scale=1, minimum-scale=1, width=device-width">
  <title>Error 404 (Not Found)!!1</title>
  <style>
    *{margin:0;padding:0}html,code{font:15px/22px arial,sans-serif}html{background:#fff;colo
  </style>
  <a href=//www.google.com/><span id=logo aria-label=Google></span></a>
  <p><b>404.</b> <ins>That’s an error.</ins>
  <p>The requested URL <code>/invalid</code> was not found on this server.  <ins>That’s all
```

1.1.3 Leveraging Proxies with Reqwest Anonymity is a cornerstone of effective web reconnaissance. Reqwest provides support for proxy configurations, allowing us to route requests through different IP addresses to further obfuscate our origin. The following example demonstrates how to configure Reqwest to use a proxy for web requests:

```
use reqwest;

async fn browse_with_proxy(url: &str, proxy: reqwest::Proxy) -> Result<(), reqwest::Error>
    let client = reqwest::Client::builder()
        .build()?;

    let response = client.get(url).send().await?;

    println!("{}", response.text().await?);

    Ok(())
}

fn main() {
    let proxy = reqwest::Proxy::all("socks5://192.168.1.1:9000")?;
    browse_with_proxy("https://www.google.com/invalid", proxy).await.unwrap();
}
```

In this example, we configure Reqwest to use a proxy, directing the requests through the specified proxy URL. This enhances anonymity by masking the original source IP address.

```
+-------------------------+
|   Proxies and Reqwest   |
|    for Web Anonymity    |
+-------------------------+
             |
             V
+-------------+-----------+
| Configure Proxy         |
|socks5://192.168.1.1:9000|
+-------------+-----------+
             |
             V
+------------------------+
|   Reqwest Client       |
|   Configure Proxy      |
|URL: socks5://192.168...|
+------------------------+
             |
             V
+------------------------+
| Make Request through   |
|   Configured Proxy     |
+------------------------+
             |
             V
+------------------------+
| Get Response via Proxy |
|  (Hidden Origin)       |
+------------------------+
             |
             V
+------------------------+
|    Print Response      |
+------------------------+
```

```rust
use reqwest;

async fn browse_with_proxy(url: &str, proxy: reqwest::Proxy) -> Result<(), reqwest::Error>
    let client = reqwest::Client::builder()
        .build()?;

    let response = client.get(url).send().await?;
    println!("{}", response.text().await?);

    Ok(())
}

let proxy = reqwest::Proxy::all("socks5://192.168.1.1:9000")?;
browse_with_proxy("https://www.google.com/invalid", proxy).await.unwrap();
```

```
<!DOCTYPE html>
<html lang=en>
  <meta charset=utf-8>
  <meta name=viewport content="initial-scale=1, minimum-scale=1, width=device-width">
  <title>Error 404 (Not Found)!!1</title>
  <style>
    *{margin:0;padding:0}html,code{font:15px/22px arial,sans-serif}html{background:#fff;colo
  </style>
  <a href=//www.google.com/><span id=logo aria-label=Google></span></a>
  <p><b>404.</b> <ins>That's an error.</ins>
  <p>The requested URL <code>/invalid</code> was not found on this server.  <ins>That's all
```

1.1.4 Building a Browser Struct To encapsulate the techniques discussed into a cohesive tool, we can construct a `Browser` struct in Rust. This struct can encapsulate the configuration of cookies, user agents, and proxies, providing a modular and reusable solution for web browsing. The following is a simplified example, and in practice, additional error handling and feature customization would be necessary:

```
use reqwest;

struct Browser {
    client: reqwest::Client,
}

impl Browser {
    fn new() -> Result<Self, reqwest::Error> {
        let client = reqwest::Client::builder()
            .cookie_store(true)
            .user_agent("Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML
            .build()?;

        Ok(Browser { client })
    }

    async fn browse(&self, url: &str) -> Result<(), reqwest::Error> {
        let response = self.client.get(url).send().await?;
        println!("{}", response.text().await?);
```

```rust
        Ok(())
    }
}


fn main() {
    let browser = Browser::new().unwrap();
    browser.browse("http://www.google.com/invalid").await.unwrap();
}
```

In this example, the Browser struct is initialized with default configurations, and the browse method is used to make requests. This encapsulation facilitates the creation of a flexible and extensible browsing tool.

```
+-------------------------+
|    Browser Struct       |
|  Modular Web Browsing   |
+-------------------------+
            |
            V
+----------------------+
|  Initialize Browser  |
|  with Default Config |
+----------------------+
            |
            V
+--------------------------+
| Reqwest Client Config    |
|   - Cookies Enabled      |
|   - User-Agent Set       |
+--------------------------+
            |
            V
+--------------------------+
|   Browser Struct         |
|    (With Configured      |
|      Reqwest Client)     |
+--------------------------+
            |
            V
+--------------------------+
|  Browse Method           |
|  Make Request and Print  |
|              Response    |
+--------------------------+
            |
             V
+--------------------------+
|     Main Program         |
|  Create Browser Instance |
|  and Perform Web Browsing|
+--------------------------+
```

```
use reqwest;

struct Browser {
    client: reqwest::Client,
}

impl Browser {
    fn new() -> Result<Self, reqwest::Error> {
        let client = reqwest::Client::builder()
            .cookie_store(true)
            .user_agent("Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML
            .build()?;

        Ok(Browser { client })
    }

    async fn browse(&self, url: &str) -> Result<(), reqwest::Error> {
        let response = self.client.get(url).send().await?;
        println!("{}", response.text().await?);
        Ok(())
    }
}

let browser = Browser::new().unwrap();
browser.browse("http://www.google.com/invalid").await.unwrap();
```

```
<!DOCTYPE html>
<html lang=en>
  <meta charset=utf-8>
  <meta name=viewport content="initial-scale=1, minimum-scale=1, width=device-width">
  <title>Error 404 (Not Found)!!1</title>
  <style>
    *{margin:0;padding:0}html,code{font:15px/22px arial,sans-serif}html{background:#fff;colo
  </style>
  <a href=//www.google.com/><span id=logo aria-label=Google></span></a>
  <p><b>404.</b> <ins>That’s an error.</ins>
  <p>The requested URL <code>/invalid</code> was not found on this server.  <ins>That’s all
```

2. Social Engineering Automation

In the world of cybersecurity, social engineering plays a crucial role in orchestrating cyber attacks. Moving beyond conventional browsing, the automation of social engineering attacks with Rust becomes a strategic manner, harnessing the language's capabilities to manipulate user behavior effectively. This section not only explores the theoretical foundations but also delves into practical applications by demonstrating how Rust can serve as a crucial tool for social engineering in specific contexts, such as interacting with DuckDuckGo and Twitter.

Before launching any social-engineering attack, obtaining comprehensive information about the target is crucial. Rust's versatility extends beyond conventional web interactions, allowing for seamless communication with external services. In particular, Rust's capabilities facilitate interacting with the DuckDuckGo API, presenting a robust avenue for information gathering. This section of exploration dives into Rust's approach to querying DuckDuckGo, illustrating how the language can be employed for collecting relevant data and gaining valuable insights. By understanding and harnessing Rust's features for social engineering automation, individuals and organizations can enhance their cybersecurity practices and fortify defenses against evolving cyber threats.

2.1 DuckDuckGo API Interaction in Rust

Rust's `reqwest` library seamlessly integrates with DuckDuckGo's API, offering a Rustic interface to interact with the search giant's wealth of information. The following example showcases a basic Rust script that queries DuckDuckGo for search results:

```rust
use reqwest;

async fn duckduckgo_search(query: &str) -> Result<(), reqwest::Error> {
    let url = format!("https://api.duckduckgo.com/?q={}&format=json", query);

    let body = reqwest::get(&url).await?.text().await?;
    println!("{}", body);

    Ok(())
}

fn main() {
    duckduckgo_search("Rust programming language").await.unwrap();
}
```

This script queries DuckDuckGo for search results related to the Rust programming language.

```rust
use reqwest;
```

```rust
async fn duckduckgo_search(query: &str) -> Result<(), reqwest::Error> {
    let url = format!("https://api.duckduckgo.com/?q={}&format=json", query);

    let body = reqwest::get(&url).await?.text().await?;
    println!("{}", body);

    Ok(())
}

duckduckgo_search("Rust programming language").await.unwrap();
```

```
{"Abstract":"Rust is a multi-paradigm, general-purpose programming language that emphasizes
```

2.1.1 Parsing DuckDuckGo Search Results Interacting with the DuckDuckGo API is just the beginning; the real power lies in parsing and extracting meaningful information from the search results. Rust's capabilities in pattern matching and data manipulation become evident in the following example, where we parse and print the titles and URLs of search results:

```rust
use reqwest;
use serde_json::Value;

async fn parse_duckduckgo_results(query: &str) -> Result<(), reqwest::Error> {
    let url = format!("https://api.duckduckgo.com/?q={}&format=json", query);

    let body = reqwest::get(&url).await?.text().await?;
    let json: Value = serde_json::from_str(&body).unwrap();

    if let Some(results) = json["RelatedTopics"].as_array() {
        for result in results {
            if let Some(text) = result["Text"].as_str() {
                println!("Result: {}", text);
                println!("---");
            }
        }
    }

    Ok(())
}

fn main() {
    parse_duckduckgo_results("Rust programming language").await.unwrap();
}
```

In this example, the script queries DuckDuckGo parses the JSON response, and prints the titles and URLs of the search results. The serde_json crate facilitates

JSON parsing in Rust.

```
:dep serde_json = { version="1.0.108" }
```

```rust
use reqwest;
use serde_json::Value;

async fn parse_duckduckgo_results(query: &str) -> Result<(), reqwest::Error> {
    let url = format!("https://api.duckduckgo.com/?q={}&format=json", query);

    let body = reqwest::get(&url).await?.text().await?;
    let json: Value = serde_json::from_str(&body).unwrap();

    if let Some(results) = json["RelatedTopics"].as_array() {
        for result in results {
            if let Some(text) = result["Text"].as_str() {
                println!("Result: {}", text);
                println!("---");
            }
        }
    }

    Ok(())
}

parse_duckduckgo_results("Rust programming language").await.unwrap();
```

```
Result: Rust (programming language) Category
---
Result: History of programming languages - The history of programming languages spans from c
---
Result: Pattern matching programming languages
---
Result: Multi-paradigm programming languages
---
Result: Statically typed programming languages
---
Result: Systems programming languages
---
Result: Concurrent programming languages
---
Result: High-level programming languages
---
Result: Mozilla
---
Result: Free software projects
---
```

```
Result: Functional languages
---
Result: Procedural programming languages
---
Result: Free compilers and interpreters
---
Result: Software using the Apache license
---
Result: Software using the MIT license
---
```

2.2 Advanced DuckDuckGo Interactions

Beyond basic search queries, Rust empowers us to undertake more advanced interactions with the DuckDuckGo API. For instance, we can explore features like image search, news search, or even querying specific websites for information. Rust's flexibility allows for the expansion of capabilities based on the requirements of the reconnaissance.

2.2.1 DuckDuckGo Image Search Expanding our Rust script to include image search functionality involves modifying the DuckDuckGo API endpoint and adapting the parsing logic. The following example demonstrates a basic Rust script for querying DuckDuckGo for image search results:

```rust
use reqwest;
use serde_json::Value;

async fn duckduckgo_image_search(query: &str) -> Result<(), reqwest::Error> {
    let url = format!("https://api.duckduckgo.com/?q={}&format=json&iax=images&ia=images", 

    let body = reqwest::get(&url).await?.text().await?;
    let json: Value = serde_json::from_str(&body).unwrap();

    if let Some(related_topics) = json["RelatedTopics"].as_array() {
        for topic in related_topics {
            if let Some(icon) = topic["Icon"].as_object() {
                if let Some(icon_url) = icon["URL"].as_str() {
                    if !icon_url.is_empty() {
                        let full_url = format!("https://duckduckgo.com{}", icon_url);
                        println!("Image URL: {}", full_url);
                        println!("---");
                    }
                }
            }
        }
    }
    Ok(())
}

fn main() {
    duckduckgo_image_search("Rust").await.unwrap();
}
```

In this example, the script queries DuckDuckGo for image search results related to the Rust programming language and prints the image URLs.

```
use reqwest;
use serde_json::Value;

async fn duckduckgo_image_search(query: &str) -> Result<(), reqwest::Error> {
    let url = format!("https://api.duckduckgo.com/?q={}&format=json&iax=images&ia=images",

    let body = reqwest::get(&url).await?.text().await?;
    let json: Value = serde_json::from_str(&body).unwrap();

    if let Some(related_topics) = json["RelatedTopics"].as_array() {
        for topic in related_topics {
            if let Some(icon) = topic["Icon"].as_object() {
                if let Some(icon_url) = icon["URL"].as_str() {
                    if !icon_url.is_empty() {
                        let full_url = format!("https://duckduckgo.com{}", icon_url);
                        println!("Image URL: {}", full_url);
                        println!("---");
                    }
                }
            }
        }
    }

    Ok(())
}

duckduckgo_image_search("Rust").await.unwrap();
```

```
Image URL: https://duckduckgo.com/i/2f16ac81.jpg
---
Image URL: https://duckduckgo.com/i/832f249b.png
---
Image URL: https://duckduckgo.com/i/playrust.com.ico
---
```

2.2.2 Customized DuckDuckGo Searches Tailoring DuckDuckGo searches to specific websites or domains enhances the precision of information retrieval. Rust's expressive syntax facilitates the creation of scripts that target particular domains or types of content. The following example illustrates a Rust script that searches for Rust-related content specifically on Wikipedia:

```rust
use reqwest;
use serde_json::Value;

async fn duckduckgo_web_search(query: &str, site: &str) -> Result<(), reqwest::Error> {
    let url = format!("https://api.duckduckgo.com/?q={}&site:{}&format=json", query, site);

    let body = reqwest::get(&url).await?.text().await?;
    let json: Value = serde_json::from_str(&body).unwrap();

    if let Some(results) = json["RelatedTopics"].as_array() {
        for result in results {
            if let (Some(title), Some(url)) = (result["Text"].as_str(), result["FirstURL"].a
                println!("Title: {}", title);
                println!("URL: {}", url);
                println!("---");
            }
        }
    }

    Ok(())
}

fn main() {
    duckduckgo_web_search("rust", "wikipedia.org").await.unwrap();
}
```

In this example, the script queries DuckDuckGo for results related to the Rust programming language but restricts the search to the Wikipedia domain. This showcases the adaptability of Rust in tailoring searches to specific contexts.

```rust
use reqwest;
use serde_json::Value;

async fn duckduckgo_web_search(query: &str, site: &str) -> Result<(), reqwest::Error> {
    let url = format!("https://api.duckduckgo.com/?q={}&site:{}&format=json", query, site);

    let body = reqwest::get(&url).await?.text().await?;
    let json: Value = serde_json::from_str(&body).unwrap();

    if let Some(results) = json["RelatedTopics"].as_array() {
        for result in results {
            if let (Some(title), Some(url)) = (result["Text"].as_str(), result["FirstURL"].a
                println!("Title: {}", title);
                println!("URL: {}", url);
                println!("---");
            }
        }
    }

    Ok(())
}

duckduckgo_web_search("rust", "wikipedia.org").await.unwrap();
```

```
Title: Rust An iron oxide, a usually reddish-brown oxide formed by the reaction of iron and
URL: https://duckduckgo.com/Rust
---
Title: Rust (programming language) A multi-paradigm, general-purpose programming language.
URL: https://duckduckgo.com/Rust_(programming_language)
---
Title: Rust (video game) A multiplayer-only survival video game developed by Facepunch Studi
URL: https://duckduckgo.com/Rust_(video_game)
---
```

2.3 X Interaction in Rust

Social media platforms are a treasure of information, and X, formerly known as Twitter, with its wealth of real-time data, becomes a prime target for social engineering reconnaissance. In this section, we explore Rust's capabilities in interacting with Twitter API, parsing tweets, and extracting valuable insights.

2.3.1 Parsing Xeets In Rust Rust's `reqwest` library seamlessly integrates with the X API, providing a gateway to the vast ocean of tweets. The following example demonstrates a Rust script that queries X for xeets containing a specific hashtag:

```
use reqwest;
use serde_json::Value;
use base64::{Engine as _, engine::{self, general_purpose}};

async fn twitter_search(hashtag: &str) -> Result<(), reqwest::Error> {
    let consumer_key = "YOUR_TWITTER_CONSUMER_KEY";
    let consumer_secret = "YOUR_TWITTER_CONSUMER_SECRET";
    let access_token = "YOUR_TWITTER_ACCESS_TOKEN";
    let access_token_secret = "YOUR_TWITTER_ACCESS_TOKEN_SECRET";

    let bearer_token = general_purpose::STANDARD.encode(&format!("{}:{}", consumer_key, con
    let auth_header = format!("Basic {}", bearer_token);

    let auth_response = reqwest::Client::new()
        .post("https://api.twitter.com/oauth2/token")
        .header("Authorization", auth_header)
        .form(&[("grant_type", "client_credentials")])
        .send()
        .await?
        .text()
        .await?;

    let auth_body: Value = serde_json::from_str(&auth_response).unwrap();
    let token = auth_body["access_token"].as_str().ok_or("Twitter API Auth Failed!").unwrap(

    let url = format!("https://api.twitter.com/2/tweets/search/recent?query=%23{}&max_resul
    let response = reqwest::Client::new()
        .get(&url)
        .header("Authorization", format!("Bearer {}", token))
        .send()
        .await?
        .text()
        .await?;

    let json: Value = serde_json::from_str(&response).unwrap();

    if let Some(data) = json["data"].as_array() {
        for tweet in data {
            if let Some(text) = tweet["text"].as_str() {
                println!("Tweet: {}", text);
                println!("---");
            }
        }
    }

    Ok(())
}


fn main() {
twitter_search("rustlang").await.unwrap();
}
```

Replace the placeholder values in the script with your actual X API credentials. This script queries X for recent tweets containing the specified hashtag and prints the text of the xeets.

```
:dep base64 = {version="0.21.5"}
```

```
use reqwest;
use serde_json::Value;
use base64::{Engine as _, engine::{self, general_purpose}};

async fn twitter_search(hashtag: &str) -> Result<(), reqwest::Error> {
    let consumer_key = "YOUR_TWITTER_CONSUMER_KEY";
    let consumer_secret = "YOUR_TWITTER_CONSUMER_SECRET";
    let access_token = "YOUR_TWITTER_ACCESS_TOKEN";
    let access_token_secret = "YOUR_TWITTER_ACCESS_TOKEN_SECRET";

    let bearer_token = general_purpose::STANDARD.encode(&format!("{}:{}", consumer_key, con
    let auth_header = format!("Basic {}", bearer_token);

    let auth_response = reqwest::Client::new()
        .post("https://api.twitter.com/oauth2/token")
        .header("Authorization", auth_header)
        .form(&[("grant_type", "client_credentials")])
        .send()
        .await?
        .text()
        .await?;

    let auth_body: Value = serde_json::from_str(&auth_response).unwrap();
    let token = auth_body["access_token"].as_str().ok_or("Twitter API Auth Failed!").unwrap(

    let url = format!("https://api.twitter.com/2/tweets/search/recent?query=%23{}&max_resul
    let response = reqwest::Client::new()
        .get(&url)
        .header("Authorization", format!("Bearer {}", token))
        .send()
        .await?
        .text()
        .await?;

    let json: Value = serde_json::from_str(&response).unwrap();
    println!("Tweet: {}", json);
    if let Some(data) = json["data"].as_array() {
        for tweet in data {
            if let Some(text) = tweet["text"].as_str() {
                println!("Tweet: {}", text);
                println!("---");
            }
        }
    }

    Ok(())
}

twitter_search("rustlang").await.unwrap();
```

```
Tweet: {"client_id":"28218887","detail":"When authenticating requests to the Twitter API v2
```

As the output suggests, we are encountering Twitter API call failures with the error message “Client Forbidden” and we are prompted to upgrade from the free plan to the basic plan. As of the recent announcement, X API v2 introduces two new endpoints, Users Search and Trends lookup, available exclusively to developers with Pro access in the X API.

2.4 Advanced Social Engineering

Social engineering extends beyond web reconnaissance, encompassing email manipulation and mass social engineering. In this section, we explore advanced Rust scripts that interact with email services, send anonymous emails, and orchestrate mass social engineering attacks.

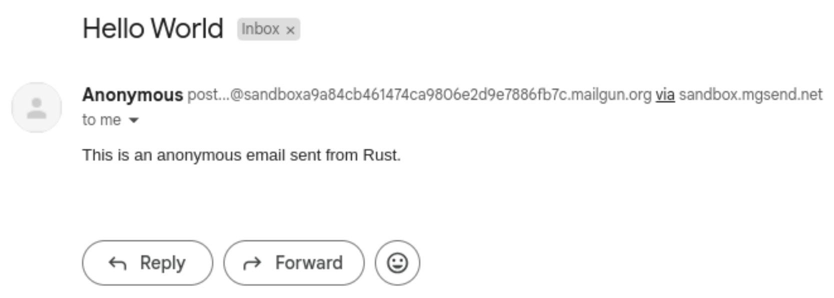
2.4.1 Anonymous Email Communication Maintaining anonymity extends beyond web browsing into email communication. Rust’s capabilities enable us to interact with email services and send anonymous emails. The following example demonstrates a Rust script that sends an anonymous email using the Reqwest library:

```
use reqwest;
use serde_json::Value;

async fn send_anonymous_email(subject: &str, body: &str) -> Result<(), reqwest::Error> {
    let sender_email = "your_sender_email@example.com";
    let receiver_email = "recipient@example.com";
    let api_key = "YOUR_EMAIL_API_KEY";

    let url = format!("https://api.mailgun.net/v3/YOUR_DOMAIN_NAME/messages");
    let client = reqwest::Client::new();

    let response = client.post(&url)
        .basic_auth("api", Some(api_key))
        .form(&[
            ("from", format!("Anonymous <{}>", sender_email)),
            ("to", receiver_email.to_string()),
            ("subject", subject.to_string()),
            ("text", body.to_string()),
        ])
        .send()
        .await?
        .text()
        .await?;


    let json: Value = serde_json::from_str(&response).unwrap();

    if let Some(message) = json["message"].as_str() {
        println!("Email Sent: {}", message);
    }

    Ok(())
}

fn main() {
    send_anonymous_email("Hello World", "This is an anonymous email sent from Rust.").await
}
```

Replace the placeholder values in the script with your actual sender email, receiver email, domain name, and **Mailgun API key**. This script sends an anonymous email using the Mailgun email service.

```
use reqwest;
use serde_json::Value;

async fn send_anonymous_email(subject: &str, body: &str) -> Result<(), reqwest::Error> {
    let sender_email = "your_sender_email@example.com";
    let receiver_email = "recipient@example.com";
    let api_key = "YOUR_EMAIL_API_KEY";

    let url = format!("https://api.mailgun.net/v3/YOUR_DOMAIN_NAME/messages");
    let client = reqwest::Client::new();

    let response = client.post(&url)
        .basic_auth("api", Some(api_key))
        .form(&[
            ("from", format!("Anonymous <{}>", sender_email)),
            ("to", receiver_email.to_string()),
            ("subject", subject.to_string()),
            ("text", body.to_string()),
        ])
        .send()
        .await?
        .text()
        .await?;

    let json: Value = serde_json::from_str(&response).unwrap();

    if let Some(message) = json["message"].as_str() {
        println!("Email Sent: {}", message);
    }

    Ok(())
}

send_anonymous_email("Hello World", "This is an anonymous email sent from Rust.").await.unwr
```

```
Email Sent: Queued. Thank you.
```


Hello World Inbox ×

Anonymous post...@sandboxa9a84cb461474ca9806e2d9e7886fb7c.mailgun.org via sandbox.mgsend.net
to me

This is an anonymous email sent from Rust.

Figure 1: Email

2.4.2 Mass Social Engineering

Mass social engineering requires automation, and Rust excels in providing the tools necessary for orchestrating large-scale attacks. The following example illustrates a Rust script that utilizes the `reqwest` library to send spear-phishing emails to multiple targets:

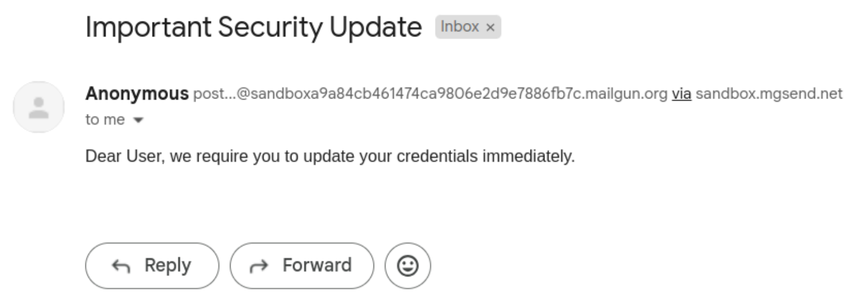
```
use reqwest;
use serde_json::Value;

async fn send_spear_phishing_email(subject: &str, body: &str, recipients: Vec<&str>) -> Resu
    let sender_email = "your_sender_email@example.com";
    let api_key = "YOUR_EMAIL_API_KEY";

    let url = format!("https://api.mailgun.net/v3/YOUR_DOMAIN_NAME/messages");
    let client = reqwest::Client::new();

    let recipient_emails = recipients.join(",");
    let response = client.post(&url)
        .basic_auth("api", Some(api_key))
        .form(&[
            ("from", format!("Anonymous <{}>", sender_email)),
            ("to", recipient_emails),
            ("subject", subject.to_string()),
            ("text", body.to_string()),
        ])
        .send()
        .await?
        .text()
        .await?;

    let json: Value = serde_json::from_str(&response).unwrap();

    if let Some(message) = json["message"].as_str() {
        println!("Email Sent: {}", message);
    }

    Ok(())
}

fn main() {
    let target_emails = ["target1@example.com", "target2@example.com", "target3@example.com
    send_spear_phishing_email("Important Security Update", "Dear User, we require you to up
}
```

Replace the placeholder values in the script with your actual **Mailgun API key**. This script utilizes the `reqwest` library to send spear-phishing emails to multiple targets, demonstrating the scalability and automation capabilities of Rust in the world of social engineering.

```
use reqwest;
use serde_json::Value;

async fn send_spear_phishing_email(subject: &str, body: &str, recipients: Vec<&str>) -> Resu
    let sender_email = "your_sender_email@example.com";
    let api_key = "YOUR_EMAIL_API_KEY";

    let url = format!("https://api.mailgun.net/v3/YOUR_DOMAIN_NAME/messages");
    let client = reqwest::Client::new();

    let recipient_emails = recipients.join(",");

    let response = client.post(&url)
```

```
        .basic_auth("api", Some(api_key))
        .form(&[
            ("from", format!("Anonymous <{}>", sender_email)),
            ("to", recipient_emails),
            ("subject", subject.to_string()),
            ("text", body.to_string()),
        ])
        .send()
        .await?
        .text()
        .await?;

    let json: Value = serde_json::from_str(&response).unwrap();

    if let Some(message) = json["message"].as_str() {
        println!("Email Sent: {}", message);
    }

    Ok(())
}

let target_emails = ["target1@example.com", "target2@example.com", "target3@example.com"];
send_spear_phishing_email("Important Security Update", "Dear User, we require you to update
```

```
Email Sent: Queued. Thank you.
```



Important Security Update

Anonymous post...@sandboxa9a84cb461474ca9806e2d9e7886fb7c.mailgun.org via sandbox.mgsend.net
to me

Dear User, we require you to update your credentials immediately.

Figure 2: Email

3. Conclusion

In this extensive exploration, we've witnessed the fusion of character and technology in the world of web reconnaissance using Rust. From anonymously browsing the Internet and interacting with DuckDuckGo and X to advanced social engineering techniques, Rust has proven to be a versatile and robust language for cybersecurity professionals. As we navigate the evolving landscape of cyber threats, Rust stands as a sweet companion, empowering us to safeguard digital landscapes with resilience.