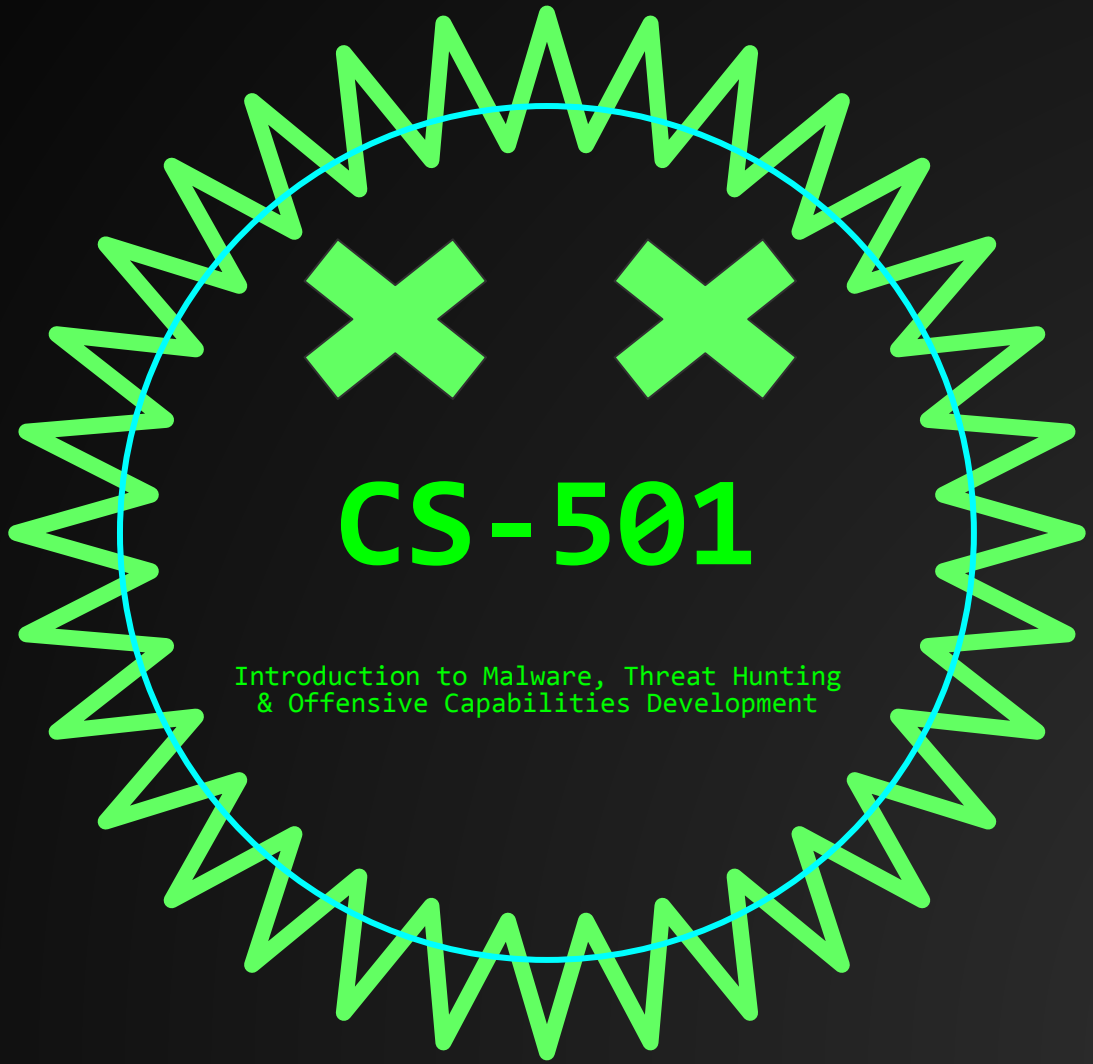

CS-501
Introduction to Malware, Threat Hunting
& Offensive Capabilities Development

Lecture 0x0c:
Capstone & C2 Engineering

Academic integrity and Mental Health

TLDR

I give incompletes.

I am not a cop.

I give extensions.

If you tell me you need an incomplete, I will give it to you no questions asked. You do not need to explain your specific situation if you don’t have the emotional energy to do so.

Submitting identical code where it is not expressly permitted is forbidden.

Capstone: Developing your own C2 Framework

To pass this class, you must develop a C2 Framework.

You may work in groups of up to 5 people.

You may not work alone. Infosec is a team sport. Go make some friends :-)

You must use a private git repository (either github or gitlab) that you add my account to. This is in part how I will observe group contributions.

You should probably get started right away.

All homework going forward is relevant to the capstone :)

Groups

I advise you to work in as large of a group as you can

There is too much work for one person to do on their own

There will be milestones your group must hit to get full credit

This is an awesome opportunity to make new friends/connections!

Milestones

- 10/31: Pick your teammates
- 10/31: Schedule your roadmap meeting with me
- 11/15: Pick your special Feature
- 11/15: Milestone one: Execution, RPC
- 11/18: Milestone two: HTTP communication and flask C2 outline
- 11/23: Milestone three: File IO
- 11/27: Milestone four: Injection
- 12/02: Milestone four: special feature
- ** tentative** 12/10: Poster session, live demo
- 12/15: Final deadline for extensions for documentation, and writeup/ last minute changes

Terminology: Review

C2: Command and Control Server

Implant: the malware “implanted” on a victim machine

Client: the software that allows the malware operators to control an implant by communicating with the team server

Sections for the Capstone

Implant

C2 Server

Client

Poster

Writeup

Requirements: Implant Functionality

For the capstone, you must implement an implant targeting the windows operating system

Your Implant needs to satisfy several requirements in order to receive full credit.

Implant Functionality Groups

- RPC and C2 Channel
- Cryptography
- Situational awareness
- Execution
- File I/O
- Persistence
- Loot
- Defense Evasion

Implant->RPC

Your implant must communicate to the C2 server using a Remote Procedure Call framework (RPC) built on top of a C2 channel and serialization format.

Example RPCs can be built using JSON, TLV, SOAP, protobuf…whatever. So long as it is communicating over an approved C2 Channel and structured data can be sent and received.

Implant->C2 Channel: RPC Tasking

- RPC must be asynchronous
- Operator uses client to create a task for an implant to execute
- Implant checks in with the server and pulls down tasks
- Implant executes tasks, and responds to the server with results. Rinse and repeat

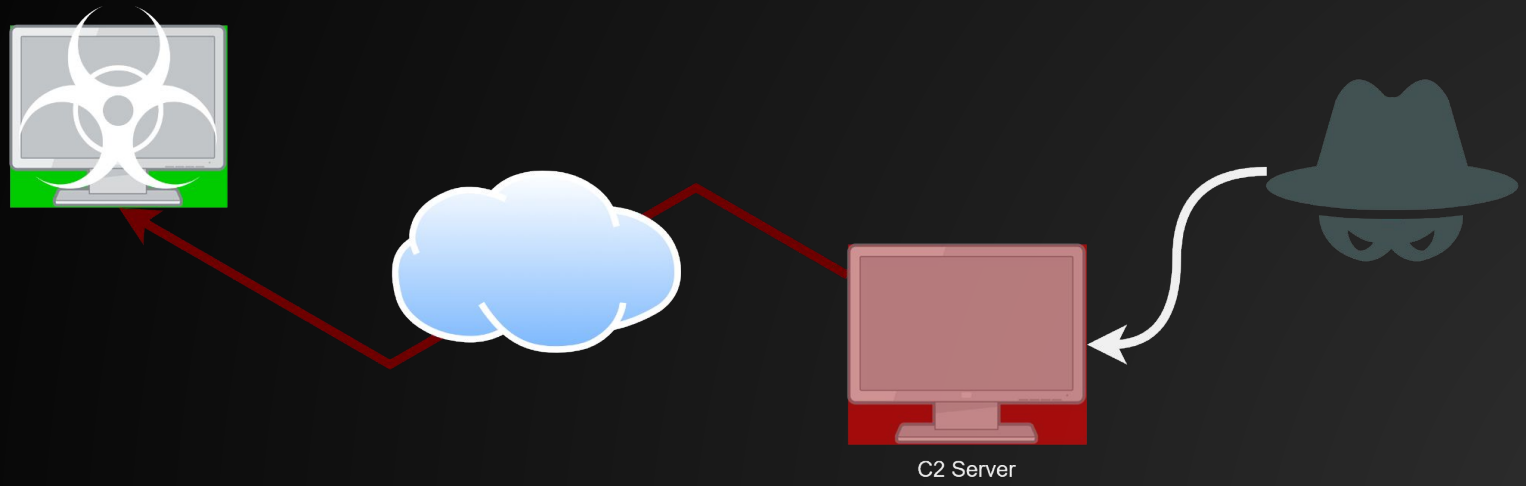
Implant->C2 Channel: HTTP

C2 Server hosts a web server, that is used to send and receive data from the implant, and the client.

Example C2 Architecture

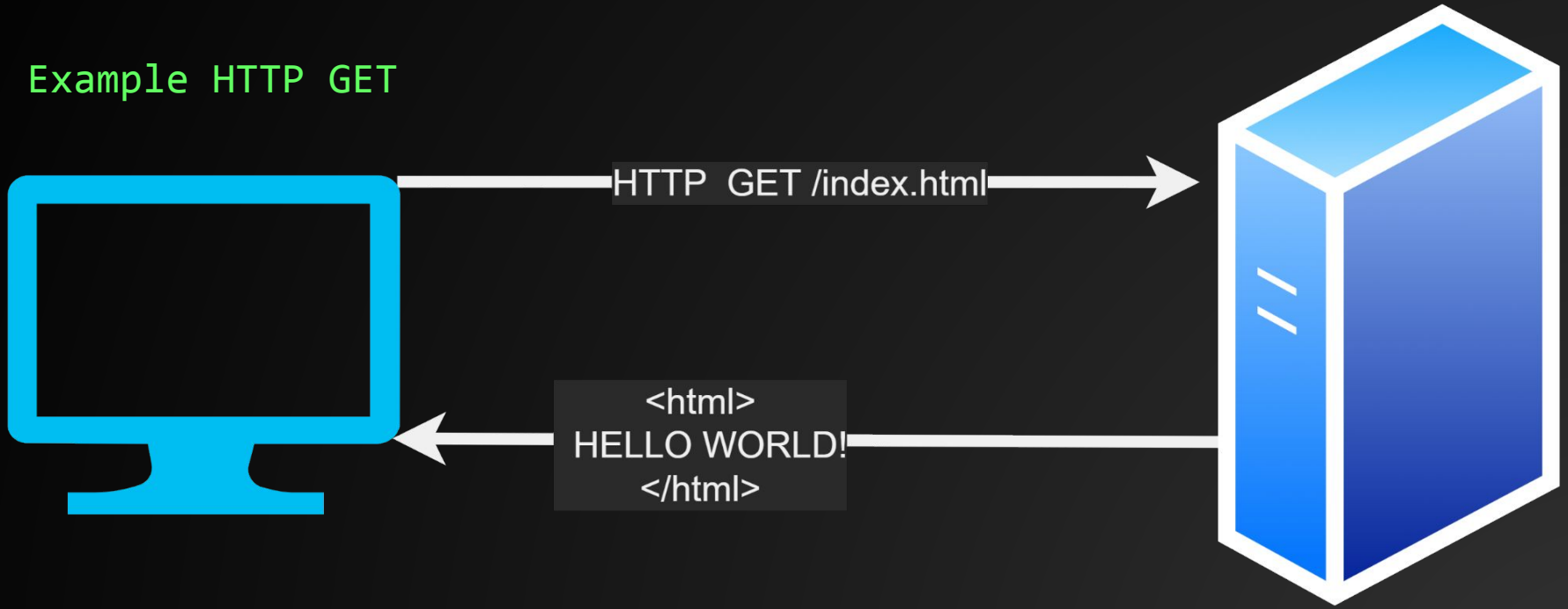
Infected Machine connects directly to the C2

Commands are issued by the operator



C2 Channel: HTTP

Example HTTP GET



HTTP RPC

RPC: Remote Procedure Call

The Malware’s RPC is the protocol used to control the malware from the server. This includes issuing commands for the malware to execute, data to upload/download..etc

Let’s consider the simple example of malware that only wishes to maintain a backdoor to a target, and execute powershell commands.

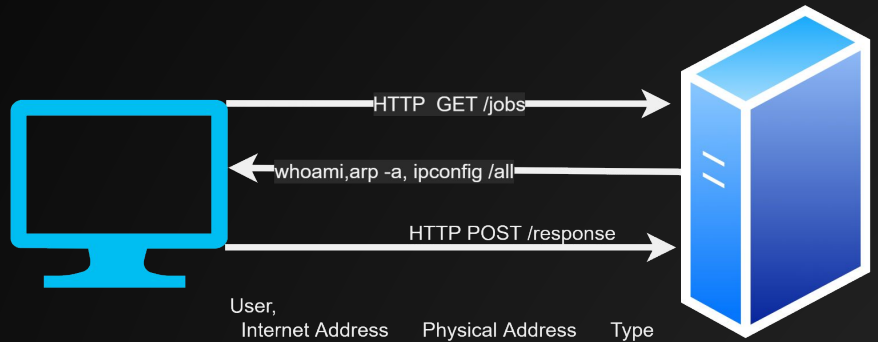
Example 1: HTTP Reverse Shell

Malware makes an HTTP GET Request to the endpoint /commands

Server responds with a list of shell commands it wishes the implant to execute

Malware responds with a post request containing the output of those commands

Example 1: HTTP Reverse Shell



User,

Internet Address	Physical Address	Type
10.16.0.1		dynamic
10.16.255.255		static
224.0.0.22		static
224.0.0.251		static
224.0.0.252		static
239.255.255.250		static
255.255.255.255		static

Connection-specific DNS Suffix . :
Description . . . . . : Intel(R) PRO/1000 MT Desktop Adapter #2
Physical Address. . . . . : 08-00-27-D6-2A-9E
DHCP Enabled. . . . . : No
Autoconfiguration Enabled . . . . : Yes
Link-local IPv6 Address . . . . . : fe80::a47d:1f11:8f29:22e6%9(Preferred)
IPv4 Address. . . . . : 10.10.10.3(Preferred)
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 10.10.10.2
DHCPv6 IAID . . . . . : 319291431
DHCPv6 Client DUID. . . . . : 00-01-00-01-28-58-BD-84-00-15-5D-24-F4-CD
DNS Servers . . . . . : 10.10.10.2
NetBIOS over Tcpip. . . . . : Enabled

HTTP Comments

- Your implant **must be proxy aware.**
  - **I.e., if the only method of egress is a web proxy, your implant must be able to discover and use it**
- You must use TLS (HTTPs)
- Your implant shouldn’t crash if internet cuts out
- Your implant does not need to be multi-threaded, but in the real world it probably does :-)
- Even if your implant uses TLS, organizations can use TLS pinning and MITM your implant…

HTTP: Config Modification

Each implant will be compiled with configuration data including

1) The C2(s) to communicate back to
2) The amount of time to go dormant before communicating again
3) The randomness to add to that sleep time
4) The kill date of the implant
5) The server’s public key

You should support modifying 1-4 at run time (no need to worry about persisting this)

Implant->Cryptography

Your implant must use **secure** cryptography.

Building blocks: Asymmetric cryptography (Public key crypto), symmetric cryptography (Ciphers/MACs), hash functions

Objectives:

- Session Establishment: Implant should establish a secure connection to a c2 server, and identify itself.
- Session update: re-establish a secure connection to a C2

Implant->Cryptography

Asymmetric:

- Implant to C2: Your implant must use asymmetric cryptography to establish a secure session to the C2.
- This could be a key exchange algorithm to agree on a key or could leverage public crypto by embedding the public key in the implant, and encrypting the session key as part of the handshake with the server.

Implant->Cryptography

Symmetric:

- Your implant must encrypt all relevant configuration, and strings. This could be something as simple as a XOR cipher, but you can get as creative as you want here. The goal of string “encryption” is to slow down the reverse engineer.
- Your implant must use authenticated crypto to communicate with the server if your C2 Channel is HTTPs. Example: AES-GCM (recommended)

Implant->Cryptography

Hash Functions:

- You must tailor your payload to a device that has the C:\malware\ch0nky.txt file path using a secure hash function. You cannot just check for the file, you must check for the hash of the file name!
- Bonus: have the stager report back the MAC address or CPU ID or other identifying information and tailor the payload to only run on a machine that it is supposed to!

Cryptography comments

Don’t roll your own crypto. Seriously, this is a bad idea, and a lot of work

Use either the Windows CNG https://docs.microsoft.com/en-us/windows/win32/seccng/cng-portal or another library.

Recommended: Monocypher, Libsodium, LibNACL, mbedtls, bearssl, LibreSSL

Not-Recommended but you do you: OpenSSL

Whatever you pick, make sure you can call it from python :-)

Implant -> Situational Awareness

Whereami, whoami, whatami…

Your implant must be able to do the following:

- Read the environment variables
- List the computer’s network interfaces
  - MAC, IPs, interface names…etc
- Get the windows version
- Get the current username and token
- Get the Computers name
- Get the Machine GUID
- List files in a directory
- Change Directory
- List all running processes
- Bonus: Anti Sandbox Detection

Implant -> Situational Awareness

Devise a way to limit the number of implants running on the same host (Mutex, named pipe...etc)

I.e., make sure that if someone clicks your binary 100 times, you still only get 1 shell

I personally like to use a tier of execution, where you limit the number of executing binaries to say 3

1 for interactive, 1 for recovery, one for long haul recovery

Implant->Execution

- CreateProcess and redirect input & output
- Shellcode Execution: support execution of shellcode in a local process
- Process Injection: support execution of shellcode in either a remote processes (by pid) or via *fork & run*
- Bonus: Integrate framework such as Donut

Implant->Execution: Payload format

Your payload must be available as

- a Portable Executable (PE)
- A Dynamic Linked Library (DLL)
- A reflective Dynamic Linked library (rDLL)
- Bonus: position independent shellcode (Donut should be able to help with this :-)

Implant->File I/O

The implant should be able to

1) Read files from the victim machine and send it back to the C2.
2) Download files from the C2 and write them to disk either **encrypted**, or **unencrypted**.

Sounds simple, turns out it is hard to do safely :-)

Safe File Upload

Spot the error

```python
@app.request("/upload_json", methods=["POST"])
def handle_upload():
    json_data = request.json
    data = json_data.get("data")
    raw_bytes = decode_data(data)
    filename = json_data.get("filename")
    loot_path = "static/" + filename
    with open(loot_path, 'wb+') as f:
        f.write(raw_bytes)
        print("I am 1337 and got wrote my loot to ", loot_path)
    return jsonify({"status": "ok"})
```

Safe File Upload

Do we actually control “filename”?

```
@app.request("/upload_json", methods=["POST"])
def handle_upload():
    json_data = request.json
    data = json_data.get("data")
    raw_bytes = decode_data(data)
    filename = json_data.get("filename")
    loot_path = "static/" + filename #This is super unsafe!
    with open(loot_path, 'wb+') as f:
        f.write(raw_bytes)
        print("I am 1337 and got wrote my loot to " , loot_path)
    return jsonify({"status": "ok"})
```

Safety

If I can successfully hack your server, with ~20 minutes of effort, you will lose 10% of your points on this.

Sanitize your paths!

https://stackoverflow.com/questions/6803505/does-my-code-prevent-directory-traversal

Implant->Persistence

You must implement functionality for the implant to persist on the target machine passed reboots.

While we strive to make all of our payloads file-less, sometimes we need to touch disk in order to persist on the machine.

The basic method of doing this is to copy the Executable to disk, and run it at startup. There are many ways to do this.

To get full credit, you must implement at least 1 strategy for persisting. Pick a fun one:
https://www.hexacorn.com/blog/2017/01/28/beyond-good-ol-run-key-all-parts/

Bonus points if you tailor your payload to the machine (i.e., the exe will only run if the machine has the same name, MAC address...etc)

Implant->Loot

Programmatically loot information from the victim machine.

- Full credit: Chrome passwords and cookies from default user profile
- Extra Credit: All chromium based passwords, cookies, autofill, and web history from all user profiles
- Or you could implement a different type of looting functionality approved by course staff. Just ask!

Implant->Defense Evasion

Required:

- Encrypt/obfuscate configuration strings (I.e. I shouldn’t be able to run strings and see your C2 channel/values used in your RPC
- No using powershell.exe or cmd.exe for situational awareness tasks

Select one of the following

1) A crypter that “packs” and obfuscates your payload. You can’t use UPX but you can use a modified version if you want!
2) A method for defeating AMSI
3) API Hashing/ dynamic IAT
4) An RPC to mimic a legitimate service
5) Payload tailoring
6) Something else of your choosing!

Opsec

Guidelines for the capstone:

- Minimal writing of files to disk
- Minimal use of cmd.exe/powershell.exe
- Minimal allocation of R/W/X memory
- Minimal creation of new processes/threads
- Separate Low latency communications from high latency actions
  - I.e., for each computer, ensure 2 payloads are run. 1 for regaining access, 1 for interactive operations
- Keep your payloads modular and small. Load functionality you need when you need it!

General Comments

Course staff will directly support students who

1) Write their C2 Server and Client in python
2) The Backend Message Broker with RabbitMQ, Redis, or ZeroMQ
3) The client UI with prompt toolkit
4) Implant in C/C++

You are welcome to use any language you want for the C2 server and client.

For the implant, your options are:

C, C++, Golang, Erlang (do it, you won't), Nim, Zig, Rust, C#, Assembly

If you use assembly for the implant, you need to comment the code a bunch.

You may not use **python for the implant.**

General Requirements: C2

Your Command and Control Server needs to be able to handle connections from multiple operators, and multiple implants.

You may accomplish this anyway you want! But course staff recommends:

1) Using Flask as the primary listener with Gunicorn as the WSGI (flask is not production ready on its own)
2) Using Postgres or MySQl as the backend Database
3) Using flask-SqlAlchemy to facilitate CRUD operations on agents, and operators
4) Using RabbitMQ/ZeroMQ/Redis to broker messages between Implant and C2, and client and C2.

C2->Database

Your database should should have several tables that correspond to different object models

1) Implants: More on next slide
2) Commands : Keep track of which operators issued what command
3) Jobs: Keep track of jobs sent to implants that are in progress/finished
4) Clients: Keep track of operators connected to the C2 via the client

C2->Database->Implants

Implant ID: Create an ID for the implant to distinguish is from others

Computer Name: What computer did it connect from?

Username: What user are you running as?

GUID: What is the computer's GUID?

Integrity: What privileges do you have?

Connecting IP address: what address did it connect from?

Session Key: After you negotiated a session key, store it per agent

Sleep: How often does the agent check in?

Jitter: How random of a check in is it?

First Seen: When did the agent first check in?

Last Seen: When was the the last time you saw the agent?

Expected Check in: When should you expect to see the agent again?

Messaging

Operators should be notified whenever one of the following occurs:

1) A new implant connects to the C2 for the first time
2) A client connects to the C2
3) An operator issues a command to an agent (current operator only)
4) An agent responds to a job that was issued by an operator

RabbitMQ Pub Sub is probably the easiest way to handle this

Your client code can be run on the same box as the C2. You may assume that you have SSH Access to your C2 server.

Another option would be to implement an endpoint to query information over HTTP

Client

Client needs to be able to securely connect to the c2, send and receive data, and get updates from the agent.

This can be a terminal interface, or a web interface

If it is a web interface, it must use authentication

Special Feature

You must implement an advanced C2 feature.

This could be an advanced

- Implant command
- RPC
- UI feature
- other

You may propose your own and I have final say over what is “advanced” :D

Special Feature implant

1) API Hashing for dynamic Resolution
2) Tailored Payloads
3) Initial Access Payload (word document, HTA, LNK...etc)
4) Defeating Impash with randomized imports
5) Token Manipulation
6) Proxy Pivots (port forwarding)
7) P2P C2 ← Automatic A
8) Custom Image Loader ← Automatic A
9) Direct System Calls ← Automatic A

Special Feature: RPC

1) Computation based sleeping
2) Advanced use of cryptography (Password Authenticated Cryptography)
3) Programmatically control your C2 via API
4) Protocol Buffers for RPC / gRPC
5) Steganography ← instant A
6) Support alternate C2 Channel: DNS, Websockets, SMTP...etc ← Instant A
7) Malleable C2: customizable RPC ← instant A

Special Feature: UI

- Use Rich and prompt toolkit to make a “pretty” User interface
- Build an HTTP panel to control the implant (with secure authentication)
- Open to suggestions!

Special Feature: other

- Trolling. I love a good troll. If your implant does something funny, that counts. But it can’t be as simple as playing music or moving the mouse. It needs to be funny.
- Containerize everything

Have an idea you think is cool? Let me know! And I will probably approve it as the special feature.

Example: troll

Steganography with a picture of me and a final payload...

Remarks

- All code needs to run on the Windows VM setup for this class
- All commands needs to be documented
- Your code needs to be coherent and commented
- Your project needs to be organized.
- Rules of thumb:
  - No functions with more than 50 lines of code
  - No files with more than 500 lines of code
  - Provide examples of usage for the client, server and implant
- Your code needs to be compilable on a Linux machine. No Visual studio unless you are writing a rootkit :D

Build Recommendations

- Stick with mingw g++ /clang++
- GNU Make is powerful enough for this project
- If you would like, you may also use cmake so long as it is documented
- For other languages/build systems, please ask me first
- You can also configure implants via loading a resource file. If you use this strategy and build implants from a template file on a linux machine, this is also OK

Build Requirements

- You need to implement a builder
- This can be source code based, but I need to be able to compile your implant with custom config for a different c2

The best C2 will get a prize :-)

The team with the “best” C2, according to course staff, will receive a mystery reward.

This will be judged on

- Opsec considerations: is your C2 easy to detect? Does it do anything that is super noisey? Does Windows Defender catch it :)
- Documentation: how easy is it to follow what your C2 does?
- How cool is the special feature?
- Is the UI usable?
- Did it make me laugh?

Projects You can, and should use for reference

https://github.com/rapid7/metasploit-payloads/tree/master/c/meterpreter (C implant)

https://github.com/bats3c/shad0w (really great C2, built by a freakin 18 year old! C + python)

https://github.com/byt3bl33d3r/SILENTTRINITY (really cool interpreter embedding, really solid python c2)

https://github.com/stephenfewer/ReflectiveDLLInjection

https://github.com/fancycode/MemoryModule (Like RDI but better)

https://github.com/TheWover/donut (turns PEs and EXEs (native and .Net) into shellcode

https://github.com/monoxgas/sRDI (Same idea(

https://docs.mythic-c2.net/ (very cool project)

https://github.com/gentilkiwi/mimikatz (Does

https://github.com/bats3c/DarkLoadLibrary (A custom implementation of LoadLibrary that works on both DLLs and reflective DLLs. Also written by the same 18 year old!)

Open Source Tools: Shad0w

- C implant, python C2
- HTTP Client
- Execution
- Defense Evasion
- Use of Cryptography
- RPC
- C2 Client

Poster requirements

- Name (or handle) for each group member
- Name of your malware
- Basic features of your malware
- Architecture diagram of how your malware works, is controlled …etc
- Analysis of operational security
- Analysis of where it could be used
- MITRE Tactics and Techniques heatmap for your malware
- At least one YARA rule/countermeasure for your malware
- IOCs associated with your malware

Demo requirements

- You need to deploy your malware in either a VM, or deploy it using a service like Heroku (recommended)
- You need to show me the malware being detonated (run), the C2 notifying operators of a new implant
- You then show me a few examples of execution (run shellcode, execute a command)
- Show me looting functionality
- Show me the special feature
- Talk about MITRE tactics and techniques used by the malware
- Talk about how to detect it

Writeup

This can be a report, or a simple README.md on your github

Must include

- Documentation of how to use your c2, what each command does..etc
- MITRE Tactics and Techniques
- Opsec considerations
- Detection rules and analysis

  Justify the design choices you made, and explain the architecture of your c2.

Timining

- I will accept late submissions for the writeup
- I will be unable accept late submissions for the presentation and the poster.
- The poster session is a requirement for passing the class.
- You need to be there.
- So far, 8 or so folks from industry have agreed to come! This is your chance to score a job/internship/make some connections!

C2 Engineering Basics

Send data

Get data

Profit

Team Server

Consider the following:

Multiple implants connect to the C2

Multiple operators connect to the C2 to control the agents

Operators need to be appraised of the activity of their colleagues and status updates of their agents

Messages are sent from the teamserver to operators

- Example messages that operators would want to see
- A new bot has connected to the server
- A bot has pulled down a task
- A bot has sent data to the server
- An operator has issued a command to an agent

Server Components and Concepts

Listeners: an HTTP listener that handles connections from implants

Database: Where we store information about clients, implants, messages...etc

Messaging: How the server communicates with clients

Databases

We use Databases to store information. Databases can be fairly simple or complex, live on disk or in memory, on one machine, or across many.

For our purposes, we are going to work with databases that exist on one machine and store structured data (i.e., it has a schema of some kind)

We already talked about an example of a database: SQLite

Non example: Document databases like MongoDB

SQL Databases: The big 4

The query language across all 4 is basically the same, with some small changes (characters for comments, built in functions, supported data types, scalability..etc)

SQLite: Lightweight, file based database. Scapable for single users but has some growing pains. Example use case: maintain state and store data for a browser

MySQL: Scalable, single machine database that is simple to manage and setup

PostgreSQL: Scalable, single machine database that is considerably more feature rich than MySQL but also more complex to manage.

Microsoft SQL: If you want to be a red teamer, you probably have to deal with Microsoft SQL at some point, but since we only care about DBs as a means to build better C2s, we don’t use it in this class.

SQL:Really, again with learning another language?

While SQL is a language, it isn’t a programming language per say. It is a way of interacting with SQL databases to perform CRUD operations (Create, Retrieve, Update, Destroy)

It is still in high demand in a professional setting, and is something I would recommend you learn for a wide variety of reasons.

Red teamers and blue teamers alike should know how databases work, as this is typically where an organization's the crown jewels live :) (Think customer databases, payment databases….etc)

SQL: Seriously oversimplified

- SQL databases are composed of tables. Tables are composed of rows. Rows are composed of data entries of a fixed type defined in the table's schema.
- In the simple case, you can think of a SQL database as an Excel Sheet, where different tabs within a sheet correspond to tables in a database.
- Each tab is filled with rows in a table, where each column has a name and type
- Data can be queried based on a standard language called SQL (Structured Query Language) to retrieve the “structured” data
- Here structured means there is a tabular schema associated with each table

Which database should you use for your C2?

Questions you need to answer first:

- How many implants do you expect to connect to your server?
- How many operators do you plan on having active at once?
- How much structured data are you collecting from implants and operators?
- What channel are you using for your C2?
- What RPC are you using for your client and implant?

You probably shouldn’t use SQLite past the POC phase

SQLite CRUD operations are blocking, and as a database can only support one “connection” at a time.

This can result in incredibly slow CRUD when there are multiple applications trying to interact with the DB

Recommendation: Stick to Postgres or MySQL

Both are production ready, incredibly powerful, and have more than enough features to handle. MySQL is probably the easier choice, but is outclassed by Postgres once the number of active connections balloons, and the schema’s complexity grows.

SQLAlchemy

- Use SQL without writing any SQL! When paired with Flask you should use Flask SQLAlchemy.
- This allows you to declare python classes instead of creating SQL schemas, and can make performing CRUD operations easy!
- Read the quickstart for a basic understanding of how to accomplish this
- https://flask-sqlalchemy.palletsprojects.com/en/2.x/quickstart

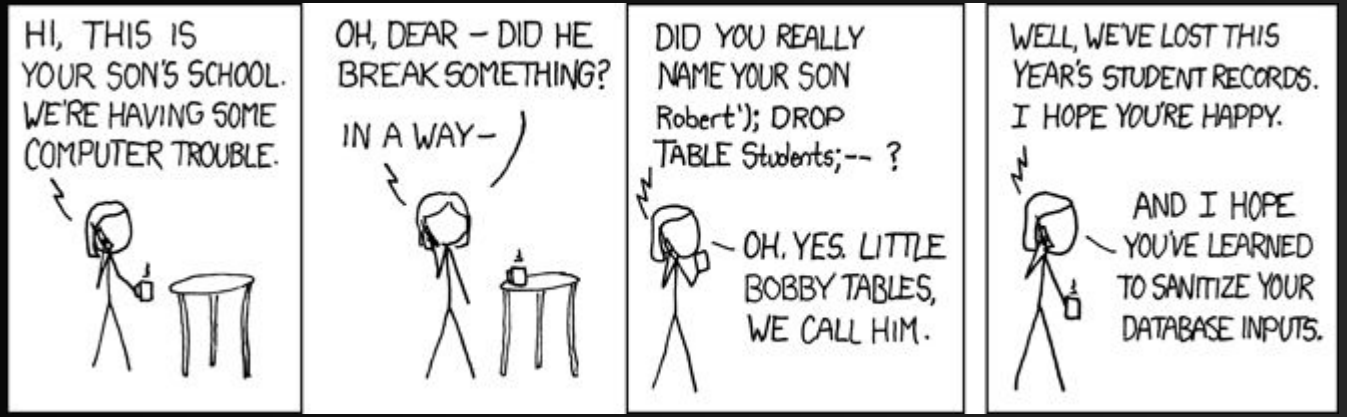
Example using Sqlite3

- All SQL DB workflows start by establishing a connection to the database
- Execute a command
- You are then given a “cursor” object that allows your to iterate through results
  - In python, a cursor is an iterable

Remarks about Safety

SQLAlchemy or other ORMs are usually safer than writing SQL queries yourself, and is also usually easier. Many malware authors shoot themselves in the foot by interacting with SQL databases in an insecure way.

Obligatory: https://xkcd.com/327/



Discussion: Drawing out a C2 framework

Flask

- Lightweight, no frills HTTP server
- Routing is handled by decorators
- Contains various helper functions to easily parse and respond to requests
- Has a rich ecosystem for different database plugins
- Is not production ready in and of itself-- requires a Web Service Gateway Interface (WSGI)

Terminology

3 programs:

Implant: the malicious backdoor

Teamserver: the server controlling the implant

Client: the client code used by the operator to control the implant

Basic Imports for our teamserver

```
from flask import Flask, request, jsonify
import multiprocessing as mp
```

Flask: flask application

request: http request object

jsonify: method to create a json response

From multiprocessing, we will use mp.Lock() to protect access to shared resources

Flask Hello world

```python
from flask import Flask

app = Flask(__name__)

@app.route("/hello")
def hello():
    return "Hello world!"
```

Setup: Server

- Server contains 4 routes
- /queue: returns the status of all of the tasks queued for the implant
- /secret: the secret endpoint to send requests to
- /tasks: endpoint for the implant to pulldown tasks from
- /response: endpoint that the implant sends the response to

Contains a shared task queue

Brokered Message Queues

An application that can be used to “broker” messages

You connect to a message broker, and it routes messages from their senders to their intended recipient in a centralized way.

BrokerLess: ZMQ

- Relies on sockets to send and receive messages
- Fewer features than rabbitmq, but is lighter and easier to manage since it is more or less a (incredibly useful!) wrapper around sockets
- One problem with ZMQ: only one thread can publish from a socket at a time. This can get annoying when you run your application in production and need to connect multiple publishers

Alternatives

Redis: Yes

Zeromq: not recommended but will work

Custom: You could, but doing so in python is going to be a pain. The GIL only allows for process based parallelism and this can get very unwieldy

SQL DB: this can technically be implemented using a database. It is a bit slow in practice

Kafka: I mean I guess?

Publish Subscribe Pattern

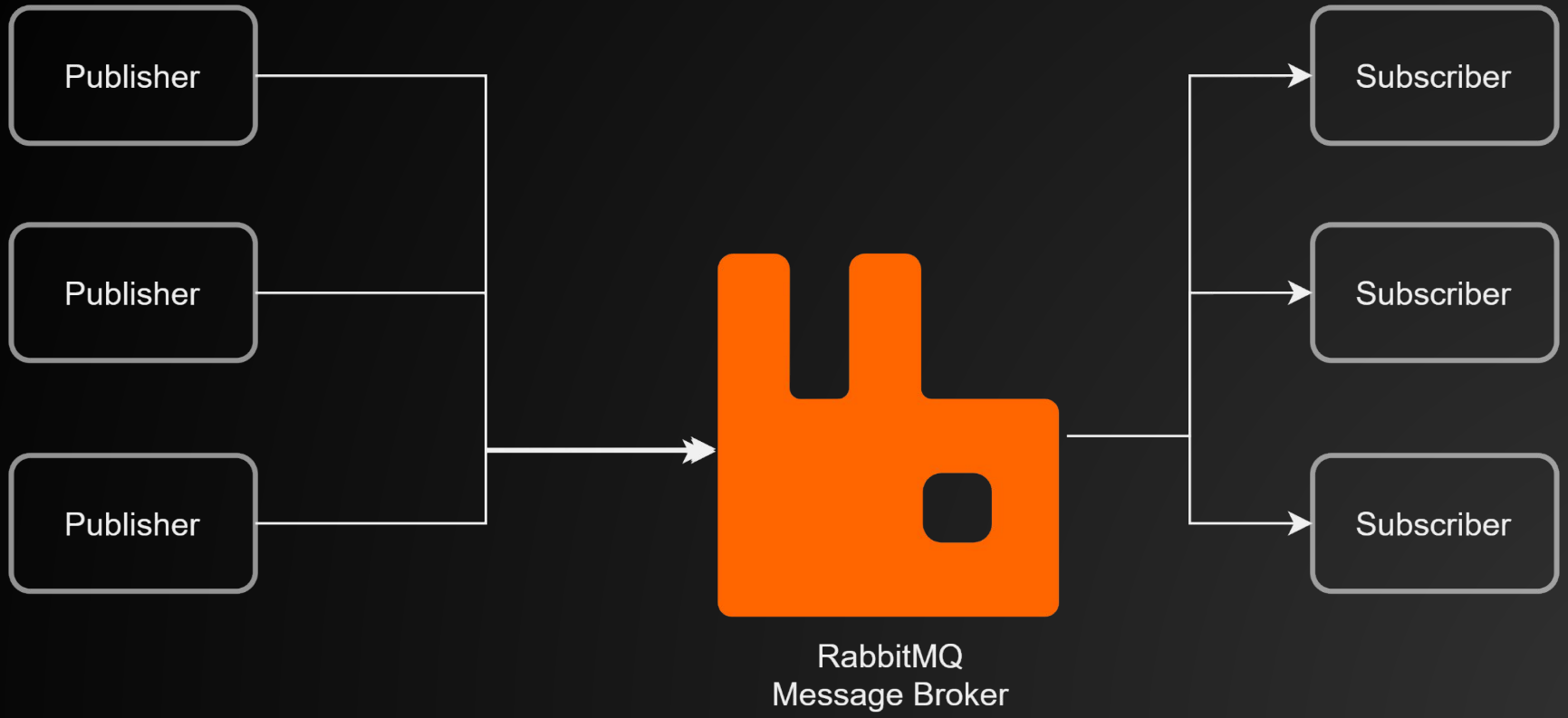
Publish Subscribe (Pub/Sub) is a messaging pattern

One or many Publishers produce messages without any knowledge where the messages will be routed. Each message has an associated topic

One or many Subscribers subscribe to a particular topic. This means they consume messages produced by the publisher

Without a message broker, you are usually limited to 1 publisher per socket.

Overview of Pub/Sub Pattern



RabbitMQ Example

```
user@LAPTOP-4RN9EIPI:~/CS-501-code-snip/pub_sub$ python3 pub.py
 [x] Sent 'Agent 0 has checked in!'
 [x] Sent 'Agent 1 has checked in!'
 [x] Sent 'Agent 2 has checked in!'
 [x] Sent 'Agent 3 has checked in!'
 [x] Sent 'Agent 4 has checked in!'
user@LAPTOP-4RN9EIPI:~/CS-501-code-snip/pub_sub$
```

```
user@LAPTOP-4RN9EIPI:~/CS-501-code-snip/pub_sub$ ipython3 sub.py
 [*] Waiting for logs. To exit press CTRL+C
 [x] b'Agent 0 has checked in!'
 [x] b'Agent 1 has checked in!'
 [x] b'Agent 2 has checked in!'
 [x] b'Agent 3 has checked in!'
 [x] b'Agent 4 has checked in!'
```

```
user@LAPTOP-4RN9EIPI:~/CS-501-code-snip/pub_sub$ ipython3 sub.py
 [*] Waiting for logs. To exit press CTRL+C
 [x] b'Agent 0 has checked in!'
 [x] b'Agent 1 has checked in!'
 [x] b'Agent 2 has checked in!'
 [x] b'Agent 3 has checked in!'
 [x] b'Agent 4 has checked in!'
```

Messaging

Messages between the operators and the server need to also be standardized.

Personally, I prefer to use JSON as it is easy to parse and serialize.

Messages are grouped into types of messages called events, which are published to operators by the server

Example events: New_implant_event, Implant_checkin_event, Implant_response_event...etc

Operators can choose which types of messages they wish to subscribe to based on a topic