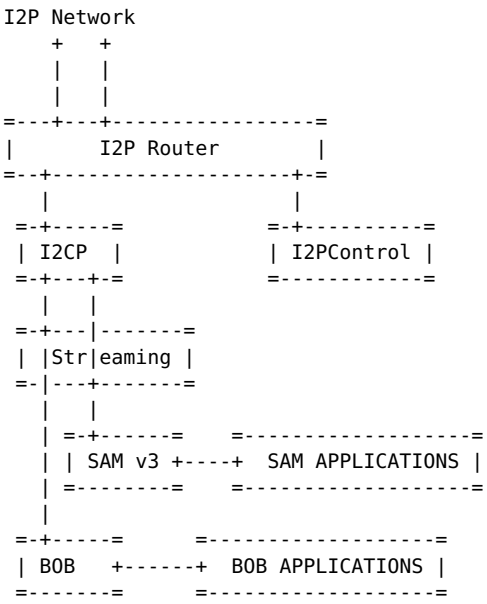
Workshop Title: I2P for Application Developers

Presenters

- idk
- I2P Application Developer
- Mostly my desk or the yard, sometimes a hammock!
- hankhill19580@gmail.com
- https://github.com/eyedeekay
- https://reddit.com/u/alreadyburnt

Current and Former I2P API's



A sketch of the I2P API's, their relationship to eachother, and their relationship to the router.

I2CP

I2CP is the core I2P API, it underlies many of the other I2P API's. It is also the most complex to use outside of Java, but recently several libraries have emerged that provide partial or full I2CP functionality from other languages. C, Java, and Javascript libraries are available, as is one Go library with partial support.

BOB

BOB is a less complicated to use API, but it is currently unmaintained. We don't plan on dropping it soon, but we also have not been adding the new features that are available in SAM Version 3. Many of the best ideas from the BOB API have been ported to the more modern SAMv3 API. Our advice is that new applications should not be using BOB, and that the applications using BOB(Both of them) should consider migrating to SAM.

SAM Version 1, 2

Deprecated and for all intents and purposes removed.

I2PControl

I2PControl is a little different. Rather than setting up connections between I2P applications, it's used for configuring and retrieving information about the router programmatically.

SAM Version 3

The current recommended API for applications of all types to communicate via I2P is the SAMv3 API. It provides a convenient way to set up, communicate through, and tear down I2P connections. It is designed so that it can implement the API familiar to your programming language in a simple and straightforward way. For example, we can implement a Socket in Java or a net.Conn in Go.

SAM will by and large be the focus of this workshop.

Which API do You Need?

If you need to make connections between applications automatically, then you need the **SAMv3 API**.

If you need to monitor or adjust the I2P router's connection, bandwidth usage, or change it's status, then you need the **I2PControl API**.

If you need to simply check the presence of an I2P router before making connections, one way is to make a quick connection to the **I2CP API**. If you're writing a Java application, the I2CP API may also be a good choice. Besides that, unless you know why you need to use I2CP, you probably just need SAM.

But Why Not Just Set Up I2PTunnel?

I2Ptunnel is good at forwarding existing services to I2P, and it can concievably be used for many applications. It does provide a SOCKS proxy after all. However, setting up i2ptunnels is an involved process, with lots of settings that are intimidating to your users. Using SAM, you set up the connections and apply all the options inside the application itself, **giving you the all-important oppourtunity to set up sane defaults** on behalf of your users.

A good example can be found in applications that are federated with Activitypub. While I2Ptunnel is perfectly capable of making AP applications available over I2P, not many new users will correctly configure the AP-based service correctly on their first try. The process of setting up connections, deciding whether or not to "Bridge" clearnet connections or remain strictly anonymous, deciding tunnel length and the number of tunnels in your destination "Pool," and most other I2P connection-related functions.

What is SAM

SAM is a simple API for controlling **connections** on the I2P i2p router in a way which is familiar to people who write internet applications. To use it, you simply set up a SAM connection and then use it like a streaming connection or to send datagrams, either with or without a repliable address. You can use these connections just like their TCP/IP equivalents for basically every intent or purpose.

Stages of the SAM Setup process

1. Handshake

- This is done so that you can negotiate the features of your SAM client with the SAM service.
- First, establish a socket connection to the I2P router's SAM Port.
- From the client, it's a simple "HELLO" message which can contain optional version and authentication information.
- When the server replies, it will respond with OK and the maximum supported SAM version.

2. Session Establishment

- Once your handshake is complete, you need to establish a session with SAM to control connections.
- To create a session, you send a "SESSION CREATE" message which must declare the type of connection and messaging you will be doing, a unique name for the connection which will allow you to refer to the client, and either a full public/private base64-encoded key pair for the local tunnel or TRANSIENT for a tunnel created with a new keypair for this session.
- Optionally, it can specify a signature type. From now on, it is recommended that libraries supporting SAM 3.1 or greater use ed25519 signatures by default.
- When the SAM service replies, it will return a result of either OK indicating that the session was established successfully or a string indicating the type of error that was encountered. If the session was established successfully, the reply will also include the destination keypair or the newly established session.

3. Connections/Messaging

- Now that you've established a session, you can start making connections and/or sending messages.
- Streaming connections are bi-directional, and can either be connected as a client to a server or listened upon to accept connections as a server to a client. Predictably, the commands you send to the SAM bridge to set up each kind of connection is "STREAM CONNECT" for connections and "STREAM ACCEPT" for listeners.
- Datagrams can be sent after a datagram style session has been established by sending datagrams to the socket. They can be repliable and include a return address or raw and not include a return address.
- Once you have created a Streaming connection, any further communication on that socket will be done with I2P, whether it be an HTTP Client, a connection between bittorrent peers, or any other kind of Streaming communication.

That all seems pretty complicated written out like that, but it's actually quite straightforward in practice and what you end up with is just a socket, and you can do whatever you would do with a regular socket. Applications that use sockets can usually simply substitute a regular socket for a SAM socket and it just works. Anything that builds on the socket can be made to build on a SAM socket, and anything that uses the SAM socketed alternative implementation can inherit I2P features.

Make your Code Re-Usable!

Because of the deliberate similarity to existing streaming and datagram communications, every language makes it possible to reduce this process to one or two steps at sensible layers of abstraction. Starting from the most similar, like a Socket in Java, a connection in Javascript, or a net.Conn in Go. The actual thing will vary from language to language, but when creating a library, you should probably start with a whatever's closest to a Socket.

Once you've done that, you've laid the foundation to alter the other network parts of your

language. In many cases, it may be possible to forward a connection using the code you've already written, or to replace an underlying structure with your SAM-enabled version.

In a surprisingly short amount of time, you too can develop extensive tooling that makes building new I2P applications and, more importantly, adapting your existing applications to use I2P simple, reliable, and familiar.

Or you can literally just write your own i2ptunnel that you can embed in your existing application. I did that once. It works really well. I don't think we need a gazillion 'socat for I2P' out there but some would argue we didn't need a third so who am I to judge.

susc

Simplest Useful SAM Streaming Client.

This isn't intended to be very "good" right now, but rather to illustrate the simplest ways the concepts of SAM map onto it's clearnet equivalents. It's been created as a set of examples for Def Con 27. When it's done being a basic example it might become a socket library, but probably not. sam3 is better.

accept.go

```
package susc

import (
    "fmt"
)

// StreamAccept asks SAM to accept a TCP-Like connection
func (c *Client) StreamAccept(id int32) (*Reply, error) {
    r, err := c.Command("STREAM ACCEPT ID=%d SILENT=false\n", id)
    if err != nil {
        return nil, err
    }

    // TODO: move check into Command()
    if r.Topic != "STREAM" || r.Type != "STATUS" {
        return nil, fmt.Errorf("Unknown Reply: %+v\n", r)
    }

    result := r.Pairs["RESULT"]
    if result != "OK" {
        return nil, fmt.Errorf("Reply Error")
    }

    return r, nil
}
```

client.go

```
package susc

import (
    "net"
        "encoding/binary"
        "encoding/base32"
    "encoding/base64"
)

type Client struct {
    *net.TCPConn
}

var (
    i2pB64enc *base64.Encoding =
base64.NewEncoding("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-~")
```

```
    i2pB32enc *base32.Encoding = base32.NewEncoding("abcdefghijklmnopqrstuvwxyz234567")
)

func NewClient() (*Client, error) {
    //var err error
    var c Client
    samaddr, err := net.ResolveTCPAddr("tcp", "127.0.0.1:7656")
    if err != nil {
        return nil, err
    }
    c.TCPConn, err = net.DialTCP("tcp", nil, samaddr)
    if err != nil {
        return nil, err
    }
    return &c, nil
}


// Base64 returns the base64 of the local tunnel
func Base64(destination string) string {
    if destination != "" {
        s, _ := i2pB64enc.DecodeString(destination)
        alen := binary.BigEndian.Uint16(s[385:387])
        return i2pB64enc.EncodeToString(s[:387+alen])
    }
    return ""
}
```

command.go

```
package susc

import (
    "bufio"
    "fmt"
)

// Command is a helper to send one command and return the reply as a string
func (c *Client) Command(str string, args ...interface{}) (*Reply, error) {
    if _, err := fmt.Fprintf(c.TCPConn, str, args...); err != nil {
        return nil, err
    }
    reader := bufio.NewReader(c.TCPConn)
    line, _, err := reader.ReadLine()
    if err != nil {
        return nil, err
    }

    return ParseReply(string(line))
}
```

connect.go

```
package susc

import (
    "fmt"
)

// StreamTCPConnect asks SAM for a TCP-Like connection to dest, has to be called on a new
Client
func (c *Client) StreamTCPConnect(id int32, dest string) error {
    r, err := c.Command("STREAM CONNECT ID=%d DESTINATION=%s\n", id, dest)
    if err != nil {
        return err
    }
```

```
    // TODO: move check into Command()
    if r.Topic != "STREAM" || r.Type != "STATUS" {
        return fmt.Errorf("Unknown Reply: %+v\n", r)
    }

    result := r.Pairs["RESULT"]
    if result != "OK" {
        return fmt.Errorf("Reply Error")
    }

    return nil
}
```

hello.go

```
package susc

import (
    "fmt"
)

func (c *Client) Hello() error {
    r, err := c.Command("HELLO VERSION MIN=3.0 MAX=3.2\n")
    if err != nil {
        return err
    }

    if r.Topic != "HELLO" {
        return fmt.Errorf("Unknown Reply: %+v\n", r)
    }

    if r.Pairs["RESULT"] != "OK" {
        return fmt.Errorf("Handshake did not succeed\nReply:%+v\n", r)
    }

    return nil
}
```

read.go

```
package susc

import (
    "bufio"
)

func (c *Client) ReadLine() (string, error) {
    reader := bufio.NewReader(c.TCPConn)
    bytes, _, err := reader.ReadLine()
    if err != nil {
        return "", err
    }
    return string(bytes), nil
}
```

reply.go

```
package susc

import (
    "fmt"
    "strings"
)

type Reply struct {
    Topic string
    Type  string
```

```
    Pairs map[string]string
}

func ParseReply(line string) (*Reply, error) {
    line = strings.TrimSpace(line)
    parts := strings.Split(line, " ")
    if len(parts) < 3 {
        return nil, fmt.Errorf("Malformed Reply.\n%s\n", line)
    }

    r := &Reply{
        Topic: parts[0],
        Type:  parts[1],
        Pairs: make(map[string]string, len(parts)-2),
    }

    for _, v := range parts[2:] {
        kvPair := strings.SplitN(v, "=", 2)
        if kvPair != nil {
            if len(kvPair) != 2 {
                return nil, fmt.Errorf("Malformed key-value-pair.\n%s\n", kvPair)
            }
        }

        r.Pairs[kvPair[0]] = kvPair[len(kvPair)-1]
    }

    return r, nil
}
```

session.go

```
package susc

import (
    "fmt"
)

func (c *Client) CreateStreamSession(id int32, dest, sigtype, options string) (string,
error) {
    if dest == "" {
        dest = "TRANSIENT"
    }
    r, err := c.Command(
        "SESSION CREATE STYLE=STREAM ID=%d DESTINATION=%s %s %s\n",
        id,
        dest,
        sigtype,
        options,
    )
    if err != nil {
        return "", err
    }

    // TODO: move check into Command()
    if r.Topic != "SESSION" || r.Type != "STATUS" {
        return "", fmt.Errorf("Unknown Reply: %+v\n", r)
    }

    result := r.Pairs["RESULT"]
    if result != "OK" {
        return "", fmt.Errorf("Reply error")
    }
    return r.Pairs["DESTINATION"], nil
}
```

write.go

```
package susc

// Write implements the TCPConn Write method.
func (c *Client) Write(b []byte) (int, error) {
    return c.TCPConn.Write(b)
}
```

Bundling an I2P Router with your SAM Application

Sometimes, the details of setting up your SAM application require you to know whether an I2P router is present and ready to accept SAM connections or not. As of release 0.9.42 in a few weeks, this becomes a very easy problem to solve. Let's take a slightly complicated case as an example, a non-JVM, non-plugin application for Windows.

Since there's a good chance your SAM Application is in a non-Java, non-JVM language, it may be difficult or impossible to build as a plugin for the I2P router. If that's the case, then we can't *assume* a router is there.

Since this is a Windows machine, we can't *assume* that a package manager is available with a viable I2P router to install. If that's the case, we'll have to install our own.

Kicking off a child installer with NSIS

One common way of creating a Windows installer for an application is to use Nullsoft Scriptable Install System. NSIS has the ability to do two essential things. First, it can check for the existence of the file, and second, it can start a new Windows application, and that application can be the I2P installer package.

```
Section "GetI2P"
  SetOutPath $INSTDIR
    IfFileExists "$PROGRAMFILES\i2p\i2p.exe" endGetI2P beginGetI2P
    Goto endGetI2P
  beginGetI2P:
    MessageBox MB_YESNO "Your system does not appear to have i2p installed.$\n$\nDo you
wish to install it now?"
    File "i2pinstaller.exe"
    ExecWait "$INSTDIR\i2pinstaller.exe"
    SetOutPath "$PROGRAMFILES\i2p"
    File "clients.config"
    SetOutPath "C:\\ProgramData\i2p"
    File "clients.config"
    SetOutPath "$AppData\I2P"
    File "clients.config"
  endGetI2P:
SectionEnd
```

As you can see, after the i2pinstaller.exe is done running, a clients.config file is copied to the I2P application data directory. We can **ONLY** do it in this case because we already determines that I2P was not installed, and it is **ONLY** in this example in this way because 0.9.42 isn't out yet.

Wait, how can I make sure the router I am bundling is current?

Well here's how I once did it in a Makefile:

```
# geti2p is an alias for i2pinstaller.exe
geti2p: i2pinstaller.exe

# This downloads the I2P installer using the url composed by the 'make url'
# target.
i2pinstaller.exe: url
    wget -c `cat geti2p.url` -O i2pinstaller.exe

# This fetches an RDF listing of I2P versions from launchpad and looks for
# the most recent stable version. Using this information, it then constructs
# a URL to download the Windows I2P router installer from Launchpad.
url:
    echo -n 'https://launchpad.net' | tee .geti2p.url
    curl -s https://launchpad.net/i2p/trunk/+rdf | \
        grep specifiedAt | \
        head -n 3 | \
        tail -n 1 | \
        sed 's|                <lp:specifiedAt rdf:resource="||g' | \
        sed 's|+rdf"/>||g' | tee -a .geti2p.url
    echo -n '+download/i2pinstall_' | tee -a .geti2p.url
```

```
    curl -s https://launchpad.net/i2p/trunk/+rdf | \
        grep specifiedAt | \
        head -n 3 | \
        tail -n 1 | \
        sed 's|                <lp:specifiedAt rdf:resource="/i2p/trunk/||g' | \
        sed 's|/+rdf"/>||g' | tee -a .geti2p.url
    echo '_windows.exe' | tee -a .geti2p.url
    cat .geti2p.url | tr -d '\n' | tee geti2p.url
    rm -f .geti2p.url
```

As we move past 0.9.42,

Wait, what if I don't want to make my clients install a JVM?

Enter Jlink, i2pd TODO

Wait, how to I finally make sure that it has the SAM API enabled?

Use clients.config.d TODO

Two big things, and one little thing, that SAM can't do and how to easily forget about them

Tell you that an I2P router is running when SAM is not enabled

Adjust the I2P Router's settings

Natively use an outproxy

Outproxy is a concept primarily known to i2ptunnel, and with i2ptunnel, primarily known to the HTTP and SOCKS proxies. SAM doesn't consider outproxying at all on it's own. If you are using SAM and want to access the clear net, you may wish to use an additional anonymizer when handling clearnet traffic, or offer a "Bridged" mode which is not anonymous but which is useful for bringing content to anonymous users

That said, since SAM connections can be used like their non-anonymous counterparts, it is actually very simple to use SAM to build an out-proxy. While this technique hasn't been used yet, and it would require careful considerations for the anonymity of the applications, there are cases where a SAM-based outproxy would be useful to incorporate into an application. One such application would be a peer-to-peer CDN which bridges I2P and clearnet content.

Examples