Workshop Title: I2P for Application Developers

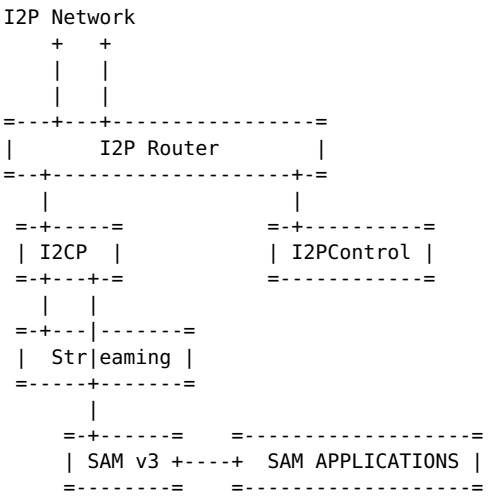
Presenters: Crypto-Privacy Village

- 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

Presenters: Monero Village

- zzz
- I2P Developer
- Location Obfuscation is a pretty big part of the point
- zzz@mail.i2p
- http://zzz.i2p

Current 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.

- https://geti2p.net/en/docs/protocol/i2cp
- https://geti2p.net/spec/i2cp

I2PTunnel

Java applications can also embed their own instances of I2PTunnel in order to use it to set up I2P tunnels. However, even non-java applications can take advantage of drop-in configuration of I2Ptunnel via i2ptunnel.config.d in the Java i2p router, or tunnels.conf.d in the C++ i2P router. In this way, regular clear-web applications can provide alternate configurations that automatically configure their I2P tunnels. If you ship i2ptunnel.config.d files, then any application can be turned into an I2P application without necessarily requiring any modification to the application's code.

- https://geti2p.net/en/docs/api/i2ptunnel

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.

- https://geti2p.net/en/docs/api/i2pcontrol

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.

- https://geti2p.net/en/docs/api/samv3

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 set up tunnels for an already-existing Web Application, then I2Ptunnel, either by embedding it in your application or by placing files into i2ptunnel.config.d.

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 Tell Users to Set Up I2PTunnel via the existing WebUI?

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 else do you need to think about?

Are you primarily:

- Publishing information or recieving it?
- Do you need ordered, Streaming-style communications
- Do you need unordered or semi-reliable Datagram-style communication
- Do you do peer discovery? How do you do peer discovery?
- When you do peer discovery, do you need to trust a service?
- Do you need to bootstrap off of a DHT?
- Do you want clients to be able to boostrap off of eachother?
- Do you want to also connect to clearnet clients?
- Do you need to be anonymous in mixed clearnet/I2P mode?
- Do you need to send datagrams in mixed clearnet/I2P mode?
- Do you want to allow people to bridge anonymous and non-anonymous clients?

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
2. Session Establishment
3. Communication

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.

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.

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.

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.

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
```

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

Two projects at least have been working on using the 'Jlink' tool to produce an I2P router that does not require the use of a JVM, those include i2p-zero from the Monero project and the Zero-Dependency installer from the I2P project. These may be preferable choices if you are bundling a router to use with a non-java application and are averse to requiring your users to install a JVM.

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

Now that we're sure an I2P router is installed, we need to make sure that a SAM API is available to your application to use. Since 0.9.42, all platforms that use SAM can also use clients.config.d. That way, you can drop a file in to an I2P router you have just installed or one that exists on the computer already, without needing to worry about over-writing or otherwise harmfully altering a potentially sensitive configuation file.

Embedding an I2P Router in your Java Application

JVM applications have another, more flexible way of working with I2P. They can use I2P as a library and selectively include and configue the components they use. This is how BiglyBT works with I2P for example.

Considerations for embedding an I2P router

While flexible, this method requires somewhat more preparation and consideration than relying on an external I2P router.

1. You will need to compile the parts of the router you want into your application.
2. You will need to periodially update your I2P router source code in order to update the i2p router embedded in your application.
3. You will need to store your configuration and some information about the network and the router.
4. You may wish to disable the Floodfill status in your embedded router. This is fine.
5. You may wish to disable participating traffic in your embedded router. In most cases, we would rather you not do this.
6. YOu should allow the user to rely on an I2P router that is already installed on the system if one is present, so the user doesn't have to effectively

For more information, see the embedding guide: http://i2p-projekt.i2p/en/docs/applications/embedding

Two big things 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

To do this effectively, you need to check for the existence of an I2CP port, but you don't need to actually use I2CP for anything else.

Adjust the I2P Router's settings

To do this, you need to use i2pcontrol.

Examples

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.

A simple TCP client

```
package susc
import (
    "encoding/base32"
    "encoding/base64"
    "encoding/binary"
    "net"
)
// Client: A SAM Client is just a socket once it's set up.
type Client struct {
    *net.TCPConn
}
// I2P uses a slightly altered base64 alphabet. You will need to customize your
// encoder to use it.
var (
    i2pB64enc *base64.Encoding =
base64.NewEncoding("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-~")
    i2pB32enc *base32.Encoding = base32.NewEncoding("abcdefghijklmnopqrstuvwxyz234567")
)
// NewClient: To create a new client, create a TCP Connection to a SAM Service.
func NewClient() (*Client, error) {
    //var err error
    var c Client
    // The default SAM address is localhost:7656
    samaddr, err := net.ResolveTCPAddr("tcp", "127.0.0.1:7656")
    if err != nil {
        return nil, err
    }
    // When you create your client, establish your connection to SAM.
    c.TCPConn, err = net.DialTCP("tcp", nil, samaddr)
    if err != nil {
        return nil, err
    }
    return &c, nil
}
// Base64 returns the base64 destination of the tunnel from the full destination.
// It's very helpful for SAM libraries to include a function like this even
// though it's not part of the spec
func Base64(destination string) string {
    if destination != "" {
        // Decode the base64 string to it's binary form
        s, _ := i2pB64enc.DecodeString(destination)
        // Take the length bits from the binary representation
        alen := binary.BigEndian.Uint16(s[385:387])
        // take the first 387 bits, + the length reported by the length bits,
        // from the binary representation and re-encode it to base64.
        return i2pB64enc.EncodeToString(s[:387+alen])
    }
    return ""
}
```

A function to send commands

```
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))
}
```

A reply parser

```
package susc
import (
    "fmt"
    "strings"
)
// Reply is a structure that represents a reply to the SAM bridge for
// convenience sake
type Reply struct {
    Topic string
    Type  string
    Pairs map[string]string
}
// ParseReply takes a string reply from the SAM bridge and turns it into a Reply
// object for later use.
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
}
```

Do the handshake

```
package susc
import (
    "fmt"
)
// Hello does the handshake with the SAM bridge
func (c *Client) Hello() error {
    reply, err := c.Command("HELLO VERSION MIN=3.0 MAX=3.2\n")
    if err != nil {
        return err
    }
    if reply.Topic != "HELLO" {
        return fmt.Errorf("Unknown Reply: %+v\n", r)
    }
    if reply.Pairs["RESULT"] != "OK" {
        return fmt.Errorf("Handshake did not succeed\nReply:%+v\n", r)
    }
    return nil
}
```

Establish a session

```
package susc
import (
    "fmt"
)
// CreateStreamSession: finally creates a streaming session. You can now use
// your socket.
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
    }
    result := r.Pairs["RESULT"]
    if result != "OK" {
        return "", fmt.Errorf("Reply error")
    }
    return r.Pairs["DESTINATION"], nil
}
```

Create a connection

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