Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

GIAC (GCIA) Gold Certification

Author: Jason Mack, jasonmack@gmail.com
Advisor: Richard Carbone
Accepted: July 15, 2015

Abstract

As the interest in collecting actionable cyber intelligence has grown substantially over the last several years in response to the growing sophistication of attackers, with it has come the need for organizations to more readily process indicators of compromise – and act immediately upon them, to determine if they are present in a given enterprise environment. While host-based tools have been designed for this very purpose, they can be challenging to deploy on an enterprise-wide basis and are dependent on frequent updates. This paper will propose several methodologies by which these indicators of compromise may be visible within network traffic. It will further study how key network security devices (e.g. Snort IDS, IPTables Firewall, Web Proxy, etc.) can be used to effectively identify and alert on indicators of compromise both on the way into the network and also via analysis of outbound traffic. In addition, STIX and TAXII will be thoroughly investigated as individual protocols, including how they can best be incorporated into the rapid generation of customized network monitoring rules.
1. Introduction

The field of cyber intelligence is a new but quickly growing discipline within Information Security. As successful attacks have become both more complex and more frequent, the need has never been greater to work collaboratively to stop malicious attackers before they are able to do harm to protected systems. One of the most effective ways to establish this collaboration has been through the creation and sharing of so-called “Indicators of Compromise” (IOC).

The OpenIOC project, an open source initiative founded by Mandiant and located at http://www.OpenIOC.org, defines an IOC as “specific artifacts left by an intrusion, or greater sets of information that allow for the detection of intrusions or other activities conducted by attackers.” (OpenIOC) Common IOCs may include hashes of known malicious files, IP addresses or DNS names of Command and Control (C&C) servers, registry keys and the contents of files (Decianno). The entire process by which IOCs are created and compared is summed up in this diagram from the OpenIOC project:

![Diagram of the IOC process](http://OpenIOC.org/resources/An_Introduction_to_OpenIOC.pdf)

Figure 1: An Introduction to OpenIOC (Source: http://OpenIOC.org/resources/An_Introduction_to_OpenIOC.pdf).

Jason Mack, jasonmack@gmail.com
In the above example, we see that the process begins with initial leads (which can also be thought of as intelligence collection), followed by IOC creation and deployment, and finally ends with a review for the presence of indicators in daily system and network activity.

Although this paper will touch upon all of these phases, the primary concentration will be on IOC deployment and identification within network traffic. In addition, although OpenIOC provides a framework to automate nearly all parts of the aforementioned process, we will be investigating the new STIX and TAXII standards – complementary but different technologies that have been adopted by NIST and MITRE for standardizing the sharing of IOCs.

However, before we can look at standardization in sharing, it is important to understand the technical basis of an IOC. Each specific IOC, be it shared via intelligence collaboration or collected internally, has a reason for its existence and a corresponding set of network technologies that would make the best choice for the implementation of detective and preventive controls. We will look at different examples of IOCs but will start with the most basic: making use of a network firewall to identify traffic from a known malicious IP address.

2. Finding Evil – Common Indicators of Compromise

2.1. IP Addresses

As mentioned, probably the most trivial IOC to work with is a simple IP address. In most modern host and network-based compromises, at least one C&C system is normally involved, although certainly the protocol it uses (HTTP, HTTPS, DNS) may vary (Qinetiq 2). Most C&C systems are deployed as part of a Botnet to send instructions to nodes known as zombies. These botnets can be extremely large, with one that was recently taken down numbering upwards of 1.9 million compromised hosts (Symantec).
Identifying IP address communication is fundamental when monitoring IPv4 based communication. Consider the following example connection to google.com, captured using Wireshark (resolved via DNS to 195.122.30.55):

![Sample connection to Google obtained using Wireshark](image)

In this example, as seen in the third column of the packet capture (after the timestamp), Google’s destination IP address can be clearly identified in the TCP/IP header. It is therefore possible to either alert or block connections to known compromised IP addresses (whether they come from open source intelligence, honeypots, or an organization’s own intelligence collection efforts) using either a firewall or an IDS/IPS.

In practice, identifying an IOC within firewall logs is a relatively simple process. For example, analyzing the network addresses of known BlackPOS Malware C&C systems that were identified in a recent community intelligence bulletin from Crowdstrike, we can use IPTables to identify and block the IOCs:

```
iptables -N LOGGING
iptables -A INPUT -j LOGGING
iptables -A LOGGING -m limit --limit 2/min -j LOG --log-prefix "IPTables Packet Dropped: " --log-level 7
iptables -A INPUT -s 199.188.204.182 -- dport 21 j DROP
iptables -A INPUT -s 50.87.167.144 -- dport 21 j DROP
iptables -A INPUT -s 63.111.113.99 -- dport 21 j DROP
```

(Source: Natarjan).

Similar rules may be implemented using the Snort IDS/IPS to alert on the presence of known C&C destination traffic:

```
iptables -N LOGGING
iptables -A INPUT -j LOGGING
iptables -A LOGGING -m limit --limit 2/min -j LOG --log-prefix "IPTables Packet Dropped: " --log-level 7
iptables -A INPUT -s 199.188.204.182 -- dport 21 j DROP
iptables -A INPUT -s 50.87.167.144 -- dport 21 j DROP
iptables -A INPUT -s 63.111.113.99 -- dport 21 j DROP
```

(Source: Natarjan).
alert tcp any any <> 199.188.204.182 21 (msg: "TargetBreach Exfil C2"; sid: xxx;)
alert tcp any any <> 50.87.167.144 21 (msg: "TargetBreach Exfil C2"; sid: xxx;)
alert tcp any any <> 63.111.113.99 21 (msg: "TargetBreach Exfil C2"; sid: xxx;)

(Source: Crowdstrike).

2.2. Domain Names and DNS

While IP addresses may be simple, a large portion of the network traffic that traverses the Internet every day is Domain Name-based. Domain names are the mapping between a URL and an IP address. When an IOC is presented as a domain name, it can be useful to both monitor it for active connections and block it in the future. In its simplest form, any host attempt to connect to a domain name will be predicated by a DNS request. The following is a simple but useful diagram of this process:

Figure 3: How Domain Name Servers Work (Source: http://computer.howstuffworks.com/dns.htm).

This diagram depicts that in order to capture potentially malicious DNS traffic, we need only capture traffic destined from client systems to DNS servers – at which point it would
then be possible to map client requests back to the domains being specifically requested. In practice, this traffic is relatively easy to identify through the use of a typical packet capture utility like Wireshark:

![Sample DNS connection traffic obtained using live testing.](image)

In this very simple example, we see a DNS query and subsequent response in packets 480 and 508 (identified by the packet ID field in the first column of each line). Packet 480 indicates that a client system (likely a web browser but it could be an application or even malware) has made a request to the local domain server for the IP address of `sans.org`. Subsequently packet 508 contains the response from the DNS server: that the IP address for `sans.org` is 66.35.59.213. As a result of analyzing this short packet capture, we can further determine that 192.168.1.104 was the client that made the original DNS request, so if `sans.org` were in fact a malicious site, incident responders could then take appropriate action now knowing that 192.168.1.104 may be infected.

Realistically, however, this is not likely the most efficient way to monitor DNS traffic for known compromised sites. Probably one of the best tools available to do this is a web proxy. In fact, “there are numerous other ways to slice and dice Web proxy logs to find bad things. For example, comparing a list of currently known malicious domains or Zeus malware domains and IPs to the proxy logs can help find hosts that have been attacked or infected, but not blocked by the Web proxy.” (Sawyer 2011)

In addition to web proxies, there are also many content filtering tools, both commercial and free that have similar functionality. A recent survey from SMB-centric community site Spiceworks provides a long list of products that stem from OpenDNS, to BlueCoat, to Squid Proxy (Spiceworks). Once a web proxy or content filtering device is deployed,
it is as simple as alerting on known bad sites or actively monitoring the logs. Log formats can differ based upon the product deployed, but will likely look something like the following:

<table>
<thead>
<tr>
<th>Source Host</th>
<th>Destination URL / Domain Name</th>
<th>Action Taken (ie: Blocked, Allowed)</th>
<th>Categorization (If content filter)</th>
</tr>
</thead>
</table>

Besides web proxies, IDS/IPS devices can also be used to look for connections to malicious domain names – by automating much of what was previously described using Wireshark. Here are sample Snort IDS rules covering two IOCs: one that alerts on any direct connection attempt using a known malicious string, and another that alerts on DNS requests to an identified malicious system:

```plaintext
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"Trojan Command And Control Request"; flow:established,to_server; uricontent:"/control_me_i_am_yours.php"; nocase; classtype:trojan-activity; sid:1233333; rev:1;)

alert udp !$DNS_SERVERS any -> $DNS_SERVERS 53 (msg:"DNS request for iamacontrolserver.ru"; content:"|01 00 00 01 00 00 00 00 00 00|"; depth:10; offset:2; content:"iamacontrolserver|02|ru"; nocase; distance:0; classtype:trojan-activity; sid:1232313; rev:1;)
```

(Source: KaffeNews).

### 2.3. HTTP Application Headers

Although IP addresses and DNS traffic both make for easy to process IOCs, they can be highly prone to false positives since by their nature they are of only limited specificity. Particularly with IP addresses, the following complaint is common:

> “Although IP addresses are a great pivot point because of their ubiquity and the amount of sources available to check against, there is one major problem with using IP addresses as a pivot point: a high rate of false-positives. The time to live (TTL) for an IP address as an effective indicator of compromise can be very low. Compromised hosts get patched, illicitly acquired hosting space is turned off, and malicious hosts are quickly identified and blocked or the traffic is black-holed by the ISP. Even when an IP address is being used for malicious activity it can

Jason Mack, jasonmack@gmail.com
sometimes be hard to block. Blocking an IP address on a shared hosting server with thousands of other legitimate sites, means also blocking all of those sites.”

(Lindka 108)

As a result, it can be useful to create and process IOCs that are more specific than just a basic IP address or domain name. These are some examples using Snort to target the same POS malware previously discussed in Section 2.1, but with specific rules for digging into the HTTP headers themselves:

```alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"MALWARE-CNC Win.Tinybanker variant outbound connection"; flow:to_server,established; content:"User-Agent: Mozilla/5.0 (compatible|3B| MSIE 9.0|3B| Windows NT 6.1|3B| Trident/5.0)|0D 0A|Content-Type: application/x-www-form-urlencoded|0D 0A|Host: "; fast_pattern:only; http_header; content:"|0D 0A|Content-Length: 13|0D 0A|Connection: Close|0D 0A|Cache-Control: no-cache|0D 0A 0D 0A|"; pcre:"/[^\x20-\x7e\n]{3}/P"; metadata:impact_flag red, policy balanced-ips drop, policy security-ips drop, ruleset community, service http; reference:url,blog.avast.com/2014/07/17/tinybanker-trojan-targets-banking-customers/; reference:url,www.virustotal.com/en/file/b88b978d00b9b3a011263f398fa6a21098aba714db14f7e71062ea4a6b2e974e/analysis/; classtype:trojan-activity; sid:31641; rev:1;)

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"MALWARE-CNC Win.Tinybanker variant outbound connection"; flow:to_server,established; urilen:4; content:"/de/"; fast_pattern:only; http_uri; content:"User-Agent: Mozilla/5.0 (compatible|3B| MSIE 9.0|3B| Windows NT 6.1|3B| Trident/5.0)|0D 0A|Content-Type: application/x-www-form-urlencoded|0D 0A|Host: "; content:"Content-Length: 13|0D 0A|Connection: Close|0D 0A|Cache-Control: no-cache|0D 0A 0D 0A|"; distance:0; metadata:impact_flag red, policy balanced-ips drop, policy security-ips drop, ruleset community, service http; reference:url,blog.avast.com/2014/07/17/tinybanker-trojan-targets-banking-customers/; reference:url,www.virustotal.com/en/file/b88b978d00b9b3a011263f398fa6a21098aba714db14f7e71062ea4a6b2e974e/analysis/; classtype:trojan-activity; sid:31642; rev:1;)
```

(Source: Crowdstrike).

Jason Mack, jasonmack@gmail.com
In these examples, we see that rather than target only a single impacted system, Snort is instead performing deep packet inspection, a task that it does in its default configuration. Dissecting the rules, alerts would occur only on outbound established connections, with specific user agents – likely to eliminate false positives for those platforms that are not affected by the suspected malware. Further packet matching occurs by inspecting the HTTP header only, and looking for specific “raw data” within the packet payload itself. Similar to an IDS/IPS, any tool that is capable of viewing and dissecting Layer 7 headers would be able to perform similar functionality. An excellent example of this might be an application firewall.

2.4. In File Contents

As mentioned at the very beginning, IOCs may include specific data in files; for example, malicious registry keys found in Windows “.reg” files. Although using network-based tools to identify a specific registry key already in place in the Windows system registry is not likely, it is quite feasible to identify this common indicator when it is in transit via IDS/IPS and through raw packet capture.

In the case of an IDS/IPS, we are able to make use of Snort’s extensive rule content-searching capabilities to look for specific strings, phrases and values. Here are a couple of good examples from the Emerging Threats rule library located at http://rules.emergingthreats.net/open/snort-2.9.0/emerging-all.rules:

```

alert tcp $SMTP_SERVERS any -> $EXTERNAL_NET 25 (msg:"ET DELETED SMTP US Confidential PROPIN"; flow:to_server,established; content:"Subject|3A|"; pcre:"/(CONFIDENTIAL|C)//[^s\w,-]*PROPIN[^s\w,-]*?=/((25)?X[1-9])/ism"; reference:url,doc.emergingthreats.net/bin/view/Main/2002447; classtype:policy-violation; sid:2002447; rev:4;)
```

Jason Mack, jasonmack@gmail.com
In the first rule, Snort alerts on browser hijack attempts by identifying any packets containing code that would modify the Windows Registry setting for Internet Explorer’s default homepage. For the second rule, Snort would trigger an alert for any email message that contained confidential markings within its body. Both of these examples are of course quite simple and may be prone to false positives (particularly the second rule), however, they do represent an overall perspective of the power of Snort for detecting file content patterns if an IOC requires it.

Another potential mechanism to accomplish the payload file matching of network traffic is a live packet capture accompanied by an appropriate content searching tool. While some packet capture applications (notably Wireshark) have the capability to search output using sophisticated search strings based on regular expressions, a better option for this task would be YARA. This is a tool that supports the creation of signatures to search the contents of files, network traffic or any other type of input for known malicious strings or IOCs. YARA is also based on Perl regular expressions with the signatures themselves contained in text files that are then processed for analysis (French). As an example, here is a YARA signature for the Scraze malware:

```yara
rule Scraze {

    strings:
        $strval1 = "C:\Windows\ScreenBlazeUpgrader.bat"
        $strval2 = "\ScreenBlaze.exe"

    condition:
        all of them
}
```

### 2.5. Specific File Hashes

Although all of the examples presented thus far are important and represent a large percentage of the network-based IOCs available in common intelligence data sets, we would be remiss if we did not address file hashes. This is because file hashes,
particularly those derived by the MD5 algorithm, are very commonly used to identify known malware samples. In many cases, IOCs start out as very simple – involving only a hash and a filename for a known bad file (Gibb 2013). Indeed, it has been this author’s personal experience that when receiving a document containing known IOCs (particularly from law enforcement sources), more often than not it will contain some sort of MD5 file hash.

A real world example of this was recently provided by the US Department of Homeland Security’s US-CERT – in a public bulletin issued to network defenders in response to a major cyber-attack: https://www.us-cert.gov/ncas/alerts/TA14-353A. This bulletin also represents a demonstrable case for why MD5 hashing continues to persist in the field of cyber threat intelligence despite the fact that SHA1 and SHA2 have long been considered cryptographically superior. While MD5 is certainly a weak hash for comparing files when integrity must be absolutely guaranteed, in the case of threat intelligence sharing the data is only as good as what is provided – as in the example here by US-CERT. Thus, as long as threat intelligence providers continue to share file hashes using MD5 and not the superior SHA1/2 algorithms (in order to ensure maximum compatibility or usability), MD5 will continue to be the hashing algorithm implemented in IOC processing.

The technical process of searching for file hashes within network traffic is not in and of itself difficult. It involves two types of tools: A packet capture program and an MD5 file calculation tool. For example, using Wireshark, we are able to analyze traffic destined to sans.org in order to identify the presence of a specific file, as shown in the following figure:

Jason Mack, jasonmack@gmail.com
By identifying and following the established TCP stream within Wireshark, we are able to recreate the file that was requested, and if needed then download it locally. This

Jason Mack, jasonmack@gmail.com
becomes much more useful when there is reason to believe that the file downloaded may be malicious because once the file is downloaded locally, without even needing to do any kind of malware analysis or reverse engineering, we can quickly compare its MD5 hash to determine if it matches any known IOCs:

![Sample execution of md5sum](image)

Figure 6: Sample execution of *md5sum* against a file obtained via Wireshark from Figure 5 obtained using live testing.

Obviously, following this process in each suspected case of malicious system communication would be highly inefficient. However, when combining this strategy of automated search using packet capture along with a recording appliance such as those from Fluke, NetScout, NetWitness or Solera – it becomes possible to extract streams such as these automatically and subsequently perform hashing and follow-up comparisons.

**3. Generating IOCs Based on Local Network Traffic**

**3.1. Background**

All of the previous IOC types have one thing in common: they are based on the processing of intelligence that was likely received from network and system data observed by others. It can be extremely useful, however, to generate IOCs based directly on anomalous activity seen on a local network. We have already discussed the types of network security devices that could be used to identify this traffic – firewalls, IDS/IPS, DNS servers, packet capture appliances and web proxy devices. This simply leaves us to determine which network activity we should be identifying that constitutes an IOC. Or more specifically, we need to ask the question: “What type of network traffic would likely represent malicious activity?”

Although this will of course be different for each environment, the information obtained from the following table is a good starting point:

**3. Generating IOCs Based on Local Network Traffic**

**3.1. Background**

All of the previous IOC types have one thing in common: they are based on the processing of intelligence that was likely received from network and system data observed by others. It can be extremely useful, however, to generate IOCs based directly on anomalous activity seen on a local network. We have already discussed the types of network security devices that could be used to identify this traffic – firewalls, IDS/IPS, DNS servers, packet capture appliances and web proxy devices. This simply leaves us to determine which network activity we should be identifying that constitutes an IOC. Or more specifically, we need to ask the question: “What type of network traffic would likely represent malicious activity?”

Although this will of course be different for each environment, the information obtained from the following table is a good starting point:
Table 1: “Top 15 Indicators of Compromise” (Source: http://www.darkreading.com/attacks-breaches/top-15-indicators-of-compromise/d/1140647?)

<table>
<thead>
<tr>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unusual Outbound Network Traffic</td>
<td>Requests to multiple sequential IP addresses, Connections to known C&amp;C systems, Requests on unusual ports</td>
</tr>
<tr>
<td>Unusual Privileged User Account Activity</td>
<td>Admin accounts logging in at odd times, accessing files they don’t normally access</td>
</tr>
<tr>
<td>Traffic from unexpected geographic locations</td>
<td>Interaction with an IP in a country that would have no legitimate reason for connectivity, logins to the same UserID from multiple geographic locations</td>
</tr>
<tr>
<td>Unusual login information</td>
<td>Failed logins for accounts that don’t exist, off hour logins, logins for multiple accounts in a short time period</td>
</tr>
<tr>
<td>Increase in volume of database accesses</td>
<td>An unusually high number of database queries – or large transfer amount</td>
</tr>
<tr>
<td>Particularly large HTML Response Sizes</td>
<td>Responses to get requests greater than a normal HTML page indicating successful SQL injection or database compromise</td>
</tr>
<tr>
<td>Large number of requests for the same file</td>
<td>Accessing the same PHP, JSP, ASP file repetitively but changing the URL string</td>
</tr>
<tr>
<td>Application requests on network ports</td>
<td>Traffic that looks like DNS on port 80, HTTP traffic on port 25, FTP over port 22</td>
</tr>
<tr>
<td>Suspicious registry or system file changes</td>
<td>Anything different from a baseline, changes to Autorun, creation of hidden files in system directories</td>
</tr>
<tr>
<td>Anomalous DNS Requests</td>
<td>A larger than normal volume of DNS requests, DNS responses without requests, DNS traffic to unusual geographic sites</td>
</tr>
<tr>
<td>Unexpected patching of systems</td>
<td>Patches applied outside of schedule</td>
</tr>
<tr>
<td>Unexplained mobile device profile change</td>
<td>Creation of new mobile device profiles, changes to existing (e.g. new certificates)</td>
</tr>
<tr>
<td>Large amounts of data in unusual locations</td>
<td>Unexplained large file archives, important files in Recycle Bin, executables in temp</td>
</tr>
<tr>
<td>Web traffic that appears to not be originating from a human requestor</td>
<td>Large volume of web traffic in a short time, unexpected browser user agent, invalid browser usage agent string</td>
</tr>
<tr>
<td>Any indication of DDOS Activity</td>
<td>Slow network or host performance, website unavailability, failover of critical devices, unusually high network load</td>
</tr>
</tbody>
</table>

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

While obviously these are only some examples of traffic that could be seen as malicious, they form a good basis for generating IOCs when reviewing local network traffic. In addition, while some of these indicators may require a host agent to optimally monitor and discover threats, they at least give the analyst an idea of what particular activities should be acted upon when seen. Typical response actions should include the implementation of blacklists, enhanced logging and monitoring or the commencement of the incident response process.

3.2. Examples

Now that we have described what constitutes an IOC, the next logical step is to be able to actually identify them, ideally in an automated fashion. Methodologies for this can range from the basic running of a *grep* command to look for suspicious patterns, to implementing a Security Incident and Event Management (SIEM) tool with correlation rules, to creating OpenIOC objects to automate the searching for each individual indicators. We will investigate each of these methods in detail.

*Grep* is a tool available by default in nearly all Unix installations, and can be downloaded and installed for free if running Windows. It provides a scriptable method to parse logs for suspicious traffic and has the capability of identifying just about anything using regular expressions. For example, we can identify files with encoding types potentially used to hide malware:

```
grep -Er "(gzinfl|base64_d)" *
```

(Source: Reilink)

While *grep* is quite simple to install and execute, full implementation of a SIEM is a much more complex process, the details of which are beyond the scope of this paper. However, using Open Source Security Information Management (OSSIM) as an example, it is possible once implemented to easily ingest indicators identified by most of the tools previously described in Section 2 and create automatic alerts. These rules are known as

Jason Mack, jasonmack@gmail.com
correlation rules, and they are essentially an XML chain of rules that query logs of network devices. So looking at two example rules prior to correlation:

```xml
<rule type="detector" name="Windows cmd.exe detected" reliability="6" time_out="60" occurrence="1" from="1:DST_IP" to="1:SRC_IP" port_from="2:DST_PORT" port_to="2:SRC_PORT" plugin_id="1001" plugin_sid="2123">

<rule type="monitor" name="Established session against abnormal port" reliability="10" from="1:SRC_IP" to="1:DST_IP" port_from="2:SRC_PORT" port_to="2:DST_PORT" plugin_id="2005" plugin_sid="248" condition="ge" value="10" interval="20" absolute="true" />

(Source: Karg)
```

Correlation is as simple as creating an XML list of rules and identifying priorities. Using the previous two examples, a correlation rule could look like this:

```xml
<rule type="detector" name="Windows cmd.exe detected"
<rule type="detector" name="Strange connection after 135/tcp or 445/tcp"
```

While SIEM rules can be very good for automatic notification and likely should be part of any IOC monitoring strategy no matter what technique is used, it is useful to have a purpose-built language for the sole purpose of creating new IOCs quickly and acting on them. In the introduction, we introduced the concept of OpenIOC and we will revisit it now as it provides an ideal methodology for creating IOCs based on live traffic. Here is an example IOC object called “Jason’s Evil Indicator” which identifies the characteristics highlighted in bold:

```xml
<?xml version="1.0" encoding="us-ascii"?>
<ioc xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmns:xsd="http://www.w3.org/2001/XMLSchema" id="23d1895c-1f57-4267-8f9d-a290b99e8f21" last-modified="2015-07-11T14:23:50"
xmns="http://schemas.mandiant.com/2010/ioc">
<short_description>Jason Mack</short_description>
<description>We saw some suspicious traffic with the following characteristics on our network:

Jason Mack, jasonmack@gmail.com
```
1. Port 80 HTTP traffic to 10.11.12.13  
2. DNS Traffic to www.definitelyevil.com  
3. User Agent is "Mozilla/5.0 (Windows; U; Windows NT 5.1; de; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3".  
4. Payload content EITHER "12596" OR "35935"

Jason Mack, jasonmack@gmail.com
Using OpenIOC as both a framework and an implementation tool, indicators can be created rapidly, and enabled for deployment to both monitoring and endpoint security devices – while also sharing with the community at large. After all, in nearly all of the previous examples, we have been referring to intelligence that likely has been derived from external sources – and the field of cyber intelligence is only as good as those organizations that are willing to share observations.

Although OpenIOC was previously referred to as a “complementary” technology to STIX & TAXII, a better description might be an “enabler” of them. The framework forms the first link in an effective, repeatable and automated chain by which IOCs can be created using OpenIOC, converted to STIX format, and then shared or deployed to network security devices using TAXII.

4. Tying it all together: STIX and TAXII

4.1. STIX

Due to their time sensitive nature, IOCs are maximally effective when they can be shared and acted upon as quickly as possible. With this goal in mind, STIX (Structured Threat Information eXpression) development began in 2010 by NIST. Its goal was to provide a common methodology and language by which all cyber threat intelligence professionals would be able to exchange data freely. The STIX format relies on what are known as “observables” or details of the specific activity that is occurring (Barnum). For example, phishing campaigns, a DDoS or a malware attack would all be examples of observables. Within each observable, STIX supports the following data elements:

Jason Mack, jasonmack@gmail.com
Table 2: STIX Standard Protocol Description (Source: official STIX project documentation at http://stixproject.github.io/data-model/1.1/).

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicator</td>
<td>Specific signs unique to the malicious activity</td>
<td>IP Addresses, DNS names, MD5 hashes</td>
</tr>
<tr>
<td>Incident</td>
<td>Systems, networks, and locations where the activity has been seen</td>
<td>External Mail Servers, Members of the G8, The SANS Institute</td>
</tr>
<tr>
<td>TTP</td>
<td>The specific tools &amp; methods being used</td>
<td>Malware signatures, exploits, source IPs, social engineering</td>
</tr>
<tr>
<td>ExploitTarget</td>
<td>Vulnerabilities or other weakness being exploited</td>
<td>CVEs, configuration issue, end users, third party vendors</td>
</tr>
<tr>
<td>Campaign</td>
<td>Motivation and reasoning for the activity</td>
<td>Cybercrime, Cyberespionage, Hacktivism</td>
</tr>
<tr>
<td>ThreatActor</td>
<td>Who is responsible for the activity</td>
<td>Org. Crime, Nation State, Inside Threat</td>
</tr>
<tr>
<td>Course of Action</td>
<td>What can be done to stop the activity</td>
<td>Apply patch, block in firewall, IDS/IPS rule</td>
</tr>
</tbody>
</table>

As our focus has been primarily IOCs, the primary STIX observable type of interest would be indicators. According to best practices, an indicator should contain at least the following items (StixProject):

- Some sort of observable to identify the malicious code or pattern that is being looked for;
- A title;
- A type (for example: malware, botnet, etc.);

Jason Mack, jasonmack@gmail.com
• Valid Time Position (The period of time for which the indicator should be considered valid);
• Indicated_TTP (A reference to a separate STIX Tactics, Techniques, and Procedures object);
• Level of confidence that the indicator is valid.

Many additional values may be part of the STIX indicator field. Some of the more interesting ones include (StixProject):
• A reference from a different source (e.g. a Snort rule or an intelligence bulletin ID);
• A “negate” value to indicate that it is actually the absence of the indicator that would be a sign of compromise;
• A description in human readable format as to what exactly the IOC is detecting;
• Which if any phases of the “Kill Chain” that this IOC is part of;
• Suggested test mechanism to verify the presence of the indicator;
• The likely impact to an individual system if the indicator were to be found;
• Suggested next steps should the indicators be found;
• Any special handling instructions;
• Notable history of sightings;
• Other related indicators or campaigns.

With this information available, it is then possible to build a STIX object that contains enough data to be both understandable and automatable. The implementation of STIX objects themselves has been largely automated by MITRE, through the creation of the STIX project located at https://github.com/STIXProject. Every major observable type as described in Table 2 has been implemented in Python – and can be easily called by any script or application that implements the STIX design standard (See Appendix 8.2 for implementation of the “Indicator” observable specifically). An example of just such a script is MITRE’s Python conversion script for converting OpenIOC objects to STIX format. As we saw previously in Section 3.2, it is possible to express live network
indicators in OpenIOC format, so when coupled with this script these indicators can be placed into a fully compliant STIX object. Appendix 8.1 contains the source code for the OpenIOC -> STIX script, and makes use of the Indicator library source code included in Appendix 8.2.

4.2. **TAXII**

TAXII was developed concurrently with STIX by the same authors and is an acronym for Trusted Automated eXchange of Indicator Information. It was developed with the goal of guaranteeing secure transport of STIX content and includes four core services: Discovery, Feed Management, Poll and Inbox. Together these services ensure the secure transfer and delivery of STIX objects as described by the following diagram (MITRE):

![Figure 6: A Description of the TAXII specification](Source: TAXII_Overview_briefing_July_2013.pdf from https://taxii.mitre.org)

The discovery service provides a list of available TAXII capable services supported by a given endpoint and when queried by a client will provide its TAXII capabilities. Feed management, as its name suggests, will provide the list of data feeds that are hosted on
the TAXII endpoint. The poll function provides the ability to request updated STIX feed data on demand. Finally, the inbox feature serves as a way to receive feed data in STIX format as it is delivered in real time (MITRE).

Just as was the case with STIX, MITRE has also provided a full Python implementation of the TAXII protocol through the development of LibTAXII (See Appendix 8.3 for most of the relevant code). This code, located at http://github.com/TAXIIProject/libtaxii, is slightly different than the previously discussed STIXProject code in that it is not necessarily designed to be a standalone implementation of the protocol. The primary reason for this lies in the fact that STIX is more of a design standard, while TAXII actually functions as the primary application for the transfer of STIX content. Putting both of these pieces together is what makes STIX & TAXII function in implementation.

4.3. **Plugging IOCs into the STIX and TAXII Infrastructure**

In addition to the development of the StixProject, MITRE has also developed support for many of the examples previously described in Section 2. For example, a Snort rule can be expressed as a STIX object using the indicator module while also including the relevant portions of the TTP and Exploit Target modules as well. Here is an example of this in practice for the recent Heartbleed vulnerability:

```markdown
```
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

Figure 7: Sample Indicator integration into a Snort Signature (Source: http://stixproject.github.io/documentation/idioms/snort-test-mechanism).

Studying this example, the author has created a STIX Observable of type “Rule” and created the required title, a confidence value of high, reference and pointers to both a related TTP and Exploit Target. It is worth noting that the author did not include a valid time position, mentioned earlier as being a STIX best practice to include.

Here is an additional example, this time embedding a firewall rule:

© 2015 The SANS Institute
Author retains full rights.
In this example, rather than using the “Indicator” module we are instead making use of the “Course of Action” module. The IOC is an IP address of 10.10.10.10 that is being associated with the PIVY C&C Server Infrastructure in the form of a TTP. As this is in fact a “course of action” and not an “indicator,” the STIX object is essentially indicating that blocking this particular TTP would serve as a way to eliminate the threat or avoid it in the first place.

Both of these examples could be either automatically created using the previously described STIX libraries, or by being converted into STIX format using a script similar to the OpenIOC -> STIX tool. Once all the necessary STIX objects have been created, we would then rely on TAXII to share them and move them to sensors and monitoring points. Currently, there are two well-known TAXII clients: Soltra Edge, located at http://www.soltra.com, and TAXII Project available at https://github.com/TAXIIProject.

Jason Mack, jasonmack@gmail.com
Soltra Edge is free to use (but not open source), while the TAXIIProject is a direct implementation by MITRE with source code available. While Soltra is functional essentially out of the box, MITRE’s solution requires a few additional components. Specifically, the Django-Taxii-Services are required in order to provide an implementation-ready instance of LibTaxii and YETI is necessary to provide the actual client functionality. YETI does not currently implement feed management, leaving it to the user to manually manage all TAXII feed subscriptions. In actuality, this is not a significant limitation however, as the number of publicly accessible, non-proprietary TAXII feeds currently available is quite limited. Just about the only current example is the Hail A Taxii project, maintained by one of the Soltra developers:

![Screenshot of Hail A Taxii](http://www.hailataxii.com)

Figure 9: Screenshot of [http://www.hailataxii.com](http://www.hailataxii.com)

Hail A Taxii aggregates content from Dshield, Emerging Threats, Phishtank, and several others. No matter which TAXII client is used, however, the process for obtaining data from the site is the same:

1. The discovery service identifies the site as TAXII compliant and establishes communication;
2. The feed management service identifies all the content available on the site and allows the user to select any or all of it for download in STIX format;
3. The poll function allows for the actual download of the feeds;
4. The inbox feature receives the STIX content, and processes it in a way that is dependent on the client itself. This can be as simple as a grouping of files in the case of YETI, or as complex as a searchable database in the case of Soltra Edge;
5. Once the STIX data has been processed through the inbox, the TAXII protocol has finished its job – and it is then the role of additional, non STIX/TAXII software to process the indicators and send them to various monitoring and defense points where they can be best used (e.g. SIEM, IDS, Firewall).

Unfortunately, even though there are two well-known TAXII clients available for use, there is still no native support by the more popular network security tools themselves – such as Snort. As a result, direct integration of STIX and TAXII based content with Snort is not possible, although this certainly does not mean it cannot be done. The easiest solution is to use one of the previously described TAXII clients alongside either automated scripts (Soltra develops several directly) or a well-established auto updater such as Oinkmaster in order to implement this functionality.

5. Limitations of Current Technology

Despite some of the strategies presented thus far, the process of identifying IOCs in network traffic is far from perfect. Primary limitations currently exist in two areas. The first lies in the capabilities and inherent weaknesses of the network security devices themselves, while the second is the lack of widespread STIX and TAXII adoption. Automated threat intelligence sharing could be vastly improved with appropriate advances in both of these areas.

As previously mentioned, a network firewall is one of the simplest devices in the network security stack – and as such, its ability to do any kind of “deep dive” for IOCs is limited. Additional functionality can be found in application firewalls that are able to more accurately inspect packet headers; however, this will limit traffic visibility to supported protocols such as HTTP and HTTPS, not to mention introduce more significant overhead. In addition, as firewalls are typically employed as detective and preventive controls

Jason Mack, jasonmack@gmail.com
between security zones, they do not have the capability to detect a system that is already compromised but that is not actively beaconing outbound traffic.

Network IDS/IPS devices suffer from a similar inherent limitation in detecting non-beaconing compromised systems. In addition, although network IDS/IPS systems such as Snort do a great job at deep packet inspection and make it possible to do deeper level packet analysis than just about any other network based technology, it still has limited capabilities when it comes to assembling very large streams. As a result, it is not able to look for specific file hashes of known malware such as a host-based system might be able to. Moreover, although its SSL preprocessor allows for the detection of anomalies that may actually be IDS evasion techniques, Snort (or any network device for that matter) would not have the capability to directly inspect SSL encrypted traffic – a common avenue for malicious activity.

Although network devices do have their technical limitations, the biggest challenge is likely the adoption of the STIX and TAXII standards themselves. While the OpenIOC project has been around for some time, STIX and TAXII as a standard has only been introduced over the last few years so adoption has only just begun. This is quite evident when one looks at the relatively minimal number of TAXII clients available – and the fact that, as previously discussed, major security products such as Snort do not provide native support. While these standards have been completely embraced by some large vendors, like any new technology, full adoption will take time.

6. Conclusion

Automated sharing of IOCs is a growing requirement. Nothing makes this statement more evident than the recent executive order instituted by President of the United States Barack Obama in response to the substantial increase in major cyber incidents:

“In order to address cyber threats to public health and safety, national security, and economic security of the United States, private companies, nonprofit organizations, executive departments and agencies (agencies), and other entities

Jason Mack, jasonmack@gmail.com
must be able to share information related to cybersecurity risks and incidents and collaborate to respond in as close to real time as possible.”

(Source: https://www.whitehouse.gov/the-press-office/2015/02/13/executive-order-promoting-private-sector-cybersecurity-information-shari)

While the President has mandated that sharing of intelligence be made easier, the primary method by which this occurs, via the collection and distribution of IOCs, are only of use if they can be acted on quickly and if the indicator is accurate from the beginning. Consequently, as threat actors continue to develop the ability to more quickly modify their TTPs, traditional IOCs become less relevant and their useful lifetime become shorter. Fortunately, with the development of fuzzy hashing, it is possible to “compare two distinctly different items and determine a fundamental level of similarity (expressed as a percentage) between the two” (Fuzzy Clarity).

However, just like STIX and TAXII, this technology is only as good as the ability of the network security devices that support it. Fuzzy hashing capabilities, as well as the more mainstream support of full file reconstruction and hash comparison within common network security systems like Snort is critical to a future where IOCs can still prove to be useful. In addition, the adoption of STIX and TAXII must also continue as it represents the best current opportunity for all sharing parties (whether they are private industry, government or individuals) to react rapidly to new threats. In an age when it seems that a new cyber event is reported every day, it is more important than ever to stay one-step ahead of the threat actors. With some improvements to the capabilities of network security devices along with the full adoption of STIX and TAXII this will become a reality.

Jason Mack, jasonmack@gmail.com
7. References


Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise


Jason Mack, jasonmack@gmail.com
8. Appendix

8.1. Python Conversion Script for OpenIOC to STIX

The following scripts is by MITRE Corporation which is free to use according to their license. It has been included here so that the reader can readily study the code to better understand the technical details of OpenIOC and STIX.

```python
# Copyright (c) 2015, The MITRE Corporation. All rights reserved.
# See LICENSE.txt for complete terms.

# OpenIOC to STIX Script
# Wraps output of OpenIOC to CybOX Script
# v0.13

import sys
import os
import traceback
import warnings
import openioc #OpenIOC Bindings
import openioc_to_cybox #OpenIOC to CybOX Script
import stix.utils
from cybox.core import Observables
from stix.indicator import Indicator
from stix.core import STIXPackage, STIXHeader

__VERSION__ = 0.13

USAGE_TEXT = """OpenIOC -- STIX Translator
v0.13 BETA // Compatible with STIX v1.1.1 and CybOX v2.1
Outputs a STIX Package with one or more STIX Indicators containing CybOX Observables translated from an input OpenIOC XML file.
Usage: python openioc_to_stix.py -i <openioc xml file> -o <stix xml file>
"""

#Print the usage text
def usage():
    print USAGE_TEXT
    sys.exit(1)

def main():
infilename = ""
outfilename = ""

#Get the command-line arguments
args = sys.argv[1:]

if len(args) < 4:
    usage()
    sys.exit(1)

for i in range(0,len(args)):
    if args[i] == '-i':
        infilename = args[i+1]
    elif args[i] == '-o':
        outfilename = args[i+1]

if os.path.isfile(infilename):
    try:
        # Perform the translation using the methods from the OpenIOC to CybOX Script
        openioc_indicators = openioc.parse(infilename)
        observables_obj = openioc_to_cybox.generate_cybox(openioc_indicators, infilename, True)
        observables_cls = Observables.from_obj(observables_obj)
    except:
        traceback.print_exc()

    # Create a STIX package with an indicator
    stix_indicators = [Indicator()]
    stix_indicators.append(Indicator(title="Example Indicator", description="This is an example indicator.", observable=observables_obj))

    stix_package = STIXPackage(header=STIXHeader(), indicators=stix_indicators)

    # Write the STIX package to file
    stix_package.write(outfilename)
```

Jason Mack, jasonmack@gmail.com
# Set the namespace to be used in the STIX Package
stix.utils.set_id_namespace({"https://github.com/STIXProject/openioc-to-stix":"openiocToSTIX")

# Wrap the created Observables in a STIX Package/Indicator
stix_package = STIXPackage()

# Add the OpenIOC namespace
input_namespaces = {"http://openioc.org/":"openioc"
stix_package.__input_namespaces__ = input_namespaces

for observable in observables_cls.observables:
    indicator_dict = {}
    producer_dict = {}
    producer_dict['tools'] = [{'name':'OpenIOC to STIX Utility', 'version':str(__VERSION__)},]
    indicator_dict['producer'] = producer_dict
    indicator_dict['title'] = "CybOX-represented Indicator Created from OpenIOC File"  
    indicator = Indicator.from_dict(indicator_dict)
    indicator.add_observable(observables_cls.observables[0])
    stix_package.add_indicator(indicator)

# Create and write the STIX Header
stix_header = STIXHeader()
stix_header.package_intent = "Indicators - Malware Artifacts"
stix_header.description = "CybOX-represented Indicators Translated from OpenIOC File"

stix_package.stix_header = stix_header

# Write the generated STIX Package as XML to the output file
outfile = open(outfilename, \'w\')

with warnings.catch_warnings():
    warnings.simplefilter(\'ignore\')
    outfile.write(stix_package.to_xml())
    warnings.resetwarnings()

outfile.flush()
outfile.close()

if __name__ == \"__main__\":
    main()

8.2. Python Source Code for the STIX “Indicator” Observable

The following scripts is by MITRE Corporation which is free to use according to their license. It has been included here so that the reader can readily study the code to better understand the technical details STIX.

# Copyright (c) 2015, The MITRE Corporation. All rights reserved.
# See LICENSE.txt for complete terms.

# external
from cybox.core import Observable, ObservableComposition
from cybox.common import Time

# internal
import stix
from stix.utils as utils
from stix.common import (Identity, InformationSource, VocabString, Confidence,
RelatedTTP, Statement, CampaignRef
)

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

from stix.common.related import (GenericRelationshipList, RelatedCOA, RelatedIndicator, RelatedCampaignRef, RelatedPackageRefs)
from stix.common.vocabs import IndicatorType
from stix.common.kill_chains import KillChainPhasesReference
import stix.bindings.indicator as indicator_binding

# relative
from .test_mechanism import TestMechanisms
from .sightings import Sightings
from .valid_time import _ValidTimePositions

class SuggestedCOAs(GenericRelationshipList):
    """The `SuggestedCOAs` class provides functionality for adding:
    :class:`stix.common.related.RelatedCOA` instances to an :class:`Indicator`
    instance.
    
    The `SuggestedCOAs` class implements methods found on
    `collections.MutableSequence` and as such can be interacted with as a
    `list` (e.g., `append()`).
    
    The `append()` method can accept instances of:
    :class:`stix.common.related.RelatedCOA` or :class:`stix.coa.CourseOfAction`
    as an argument.
    
    Note:
    Calling `append()` with an instance of
    :class:`stix.coa.CourseOfAction` will wrap that instance in a
    :class:`stix.common.related.RelatedCOA` layer, with the `item` set to
    the :class:`stix.coa.CourseOfAction` instance.
    
    Examples:
    Append an instance of :class:`stix.coa.CourseOfAction` to the
    `Indicator.suggested_coas` property. The instance of
    :class:`stix.coa.CourseOfAction` will be wrapped in an instance of
    :class:`stix.common.related.RelatedCOA`.
    >>> coa = CourseOfAction()
    >>> indicator = Indicator()
    >>> indicator.suggested_coas.append(coa)
    >>> print type(indicator.suggested_coas[0])
    <class 'stix.common.related.RelatedCOA'>
    
    Iterate over the `suggested_coas` property of an :class:`Indicator`
    instance and print the ids of each underlying
    :class:`stix.coa.CourseOfAction` instance.
    >>> for related_coa in indicator.suggested_coas:
    >>>     print related_coa.item.id_

    Args:
    suggested_coas(list): A list of :class:`stix.coa.CourseOfAction`
    or :class:`stix.common.related.RelatedCOA` instances.
    scope (str): The scope of the items. Can be set to "inclusive"
    or "exclusive". See :class:`stix.common.related.GenericRelationshipList`
    documentation for more information.
    
    Attributes:
    scope (str): The scope of the items. Can be set to "inclusive"
    or "exclusive". See :class:`stix.common.related.GenericRelationshipList`
    documentation for more information.
    
    """
    _namespace = "http://stix.mitre.org/Indicator-2"
    _binding = indicator_binding

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

_binding_class = indicator_binding.SuggestedCOAsType
_binding_var = "Suggested_COA"
_contained_type = RelatedCOA
_inner_name = "suggested_coas"

def __init__(self, suggested_coas=None, scope=None):
    super(SuggestedCOAs, self).__init__(scope, suggested_coas)

class RelatedIndicators(GenericRelationshipList):
    
    **``The RelatedIndicators`` class provides functionality for adding
    :class:`stix.common.related.RelatedIndicator` instances to an
    :class:`Indicator` instance.**

    The ``RelatedIndicators`` class implements methods found on
    `collections.MutableSequence` and as such can be interacted with as a
    `list` (e.g., `append()`).

    The `append()` method can accept instances of
    :class:`stix.common.related.RelatedIndicator` or
    :class:`Indicator` as an argument.

    **Note:**
    Calling `append()` with an instance of
    :class:`stix.coa.CourseOfAction` will wrap that instance in a
    :class:`stix.common.related.RelatedIndicator` layer, with `item`
    set to the :class:`Indicator` instance.

    **Examples:**
    Append an instance of :class:`Indicator` to the
    `Indicator.related_indicators` property. The instance of
    :class:`Indicator` will be wrapped in an instance of
    :class:`stix.common.related.RelatedIndicator`:

    >>> related = Indicator()
    >>> parent_indicator = Indicator()
    >>> parent_indicator.related_indicators.append(related)
    >>> print type(indicator.related_indicators[0])
    <class 'stix.common.related.RelatedIndicator'>

    Iterate over the `related_indicators` property of an
    :class:`Indicator` instance and print the ids of each underlying
    :class:`Indicator` instance:

    >>> for related in indicator.related_indicators:
    >>>     print related.item.id_

    **Args:**
    related_indicators (list, optional): A list of :class:`Indicator` or
    :class:`stix.common.related.RelatedIndicator` instances.
    scope (str, optional): The scope of the items. Can be set to
    `"inclusive"` or `"exclusive"`. See
    :class:`stix.common.related.GenericRelationshipList` documentation
    for more information.

    **Attributes:**
    scope (str): The scope of the items. Can be set to `"inclusive"`
    or `"exclusive"`. See
    :class:`stix.common.related.GenericRelationshipList` documentation
    for more information.

    
    _namespace = "http://stix.mitre.org/Indicator-2"
    _binding = indicator_binding
    _binding_class = indicator_binding.RelatedIndicatorsType
    _binding_var = "Related_Indicator"
    _contained_type = RelatedIndicator
    _inner_name = "related_indicators"
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

def __init__(self, related_indicators=None, scope=None):
    super(RelatedIndicators, self).__init__(scope, related_indicators)

class Indicator(stix.BaseCoreComponent):
    """Implementation of the STIX Indicator."
    Args:
    id_ (optional): An identifier. If `None`, a value will be generated
        via `mixbox.idgen.create_id()`. If set, this will unset the
        `idref` property.
    idref (optional): An identifier reference. If set this will unset the
        `id_` property.
    title (optional): A string title.
    timestamp (optional): A timestamp value. Can be an instance of
        `datetime.datetime` or `str`.
    description (optional): A string description.
    short_description (optional): A string short description.
    """
    _binding = indicator_binding
    _binding_class = indicator_binding.IndicatorType
    _namespace = 'http://stix.mitre.org/Indicator-2'
    _version = "2.2"
    _ALL_VERSIONS = ('2.0', '2.0.1', '2.1', '2.1.1', '2.2')
    _ALLOWED_COMPOSITION_OPERATORS = ('AND', 'OR')
    _ID_PREFIX = "indicator"

def __init__(self, id_=None, idref=None, timestamp=None, title=None,
             description=None, short_description=None):
    super(Indicator, self).__init__(
        id_=id_,
        idref=idref,
        timestamp=timestamp,
        title=title,
        description=description,
        short_description=short_description
    )

    self.producer = None
    self.observables = None
    self.indicator_types = IndicatorTypes()
    self.confidence = None
    self.indicatedttps = _IndicatedTTPs()
    self.testmechanisms = TestMechanisms()
    self.alternative_id = None
    self.suggestedcoas = SuggestedCOAs()
    self.sightings = Sightings()
    self.composite_indicator_expression = None
    self.killchainphases = KillChainPhasesReference()
    self.validtimepositions = _ValidTimePositions()
    self.relatedindicators = None
    self.relatedcampaigns = RelatedCampaignRefs()
    self.observablecompositionoperator = "OR"
    self.likelyimpact = None
    self.negate = None
    self.relatedpackages = RelatedPackageRefs()

@property
def producer(self):
    """Contains information about the source of the :class:`Indicator`.
    
    Default Value: `None`
    """
    return

    Returns:
    An instance of
    :class:`stix.common.information_source.InformationSource`
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

Raises:
ValueError: If set to a value that is not "None" and not an instance of:

class: stix.common.information_source.InformationSource

====
return self._producer

@producer.setter
def producer(self, value):
    self._set_var(InformationSource, try_cast=False, producer=value)

@property
def observable(self):
    """A convenience property for accessing or setting the only `cybox.core.Observable` instance held by this Indicator.

Default Value: Empty `list`.

Setting this property results in the `observables` property being reinitialized to an empty `list` and appending the input value, resulting in a `list` containing one value.

Note:
If the `observables` list contains more than one item, this property will only return the first item in the list.

Returns:
An instance of `cybox.core.Observable`.

Raises:
ValueError: If set to a value that cannot be converted to an instance of `cybox.core.Observable`.

====
if self.observables:
    return self.observables[0]
else:
    return None

@observable.setter
def observable(self, observable):
    self._observables = _Observables(observable)

@property
def observables(self):
    """A list of `cybox.core.Observable` instances. This can be set to a single object instance or a list of objects.

Note:
If the input value or values are not instance(s) of `cybox.core.Observable`, an attempt will be made to convert the value to an instance of `cybox.core.Observable`.

Default Value: Empty `list`.

Returns:
A `list` of `cybox.core.Observable` instances.

Raises:
ValueError: If set to a value that cannot be converted to an instance of `cybox.core.Observable`.

====
return self._observables

@observables.setter
def observables(self, value):

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
self._observables = _Observables(value)

def add_observable(self, observable):
    """Adds an observable to the 'observables' list property of the
class: Indicator'.

    If the 'observable' parameter is 'None', no item will be added
to the 'observables' list.

Note:
The STIX Language dictates that an class: Indicator can have only
one 'Observable' under it. Because of this, the 'to_xml()' method will convert the 'observables' list into an
'cybox.core.ObservableComposition' instance, in which each item
in the 'observables' list will be added to the composition. By
default, the 'operator' of the composition layer will be set to
''OR'' . The 'operator' value can be changed via the
'observable_composition_operator' property.

Args:
    observable: An instance of 'cybox.core.Observable' or an object
type that can be converted into one.

Raises:
    ValueError: If the 'observable' param cannot be converted into an
instance of 'cybox.core.Observable'.
    """
    self.observables.append(observable)

@property
def alternative_id(self):
    """An alternative identifier for this class: Indicator'

    This property can be set to a single string identifier or a list of
identifiers. If set to a single object, the object will be inserted
into an empty list internally.

    Default Value: Empty 'list'

    Returns:
    A list of alternative ids.
    """
    return self._alternative_id

@alternative_id.setter
def alternative_id(self, value):
    self._alternative_id = []
    if not value:
        return
cif utils.is_sequence(value):
        self._alternative_id.extend(x for x in value if x)
else:
    self._alternative_id.append(value)

def add_alternative_id(self, value):
    """Adds an alternative id to the 'alternative_id' list property.

    Note:
    If 'None' is passed in no value is added to the
    'alternative_id' list property.

    Args:
    value: An identifier value.
    """
    if not value:
```

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com
self._indicator_types = IndicatorTypes(value)

def add_indicator_type(self, value):
    """Adds a value to the `indicator_types` list property.

    The `value` parameter can be a `str` or an instance of
    `stix.common.vocabs.VocabString`.

    Note:
    If the `value` parameter is a `str` instance, an attempt will be
    made to convert it into an instance of
    `stix.common.vocabs.IndicatorType`

    Args:
    value: An instance of `stix.common.vocabs.VocabString`
    or `str`.

    Raises:
    ValueError: If the `value` param is a `str` instance that cannot
    be converted into an instance of
    `stix.common.vocabs.IndicatorType`.
    """
    self.indicator_types.append(value)

@property
def confidence(self):
    """The confidence for this :class:`Indicator`.

    This property can be set to an instance of `str`,
    `stix.common.vocabs.VocabString`, or
    `stix.common.confidence.Confidence`.

    Default Value: `None`

    Note:
    If set to an instance of `str` or
    `stix.common.vocabs.VocabString`, that value will be wrapped
    in an instance of
    `stix.common.confidence.Confidence`.

    Returns:
    An instance of
    `stix.common.confidence.Confidence`.

    Raises:
    ValueError: If set to a `str` value that cannot be converted into
    an instance of `stix.common.confidence.Confidence`.
    """
    return self._confidence

@confidence.setter
def confidence(self, value):
    self._set_var(Confidence, confidence=value)

@property
def indicated_ttps(self):
    return self._indicated_ttps

@indicated_ttps.setter
def indicated_ttps(self, value):
    self._indicated_ttps = _IndicatedTTPs(value)

def add_indicated_ttp(self, v):
    """Adds an Indicated TTP to the `indicated_ttps` list property
    of this :class:`Indicator`.

    The `v` parameter must be an instance of
    `stix.common.related.RelatedTTP` or `stix.ttp.TTP`.

    Args:
    value: An instance of `stix.common.vocabs.VocabString`
    or `str`.

    Raises:
    ValueError: If the `value` param is a `str` instance that cannot
    be converted into an instance of
    `stix.common.vocabs.IndicatorType`.
    """
    self.indicator_types.append(value)
If the 'v' parameter is 'None', no item will be added to the 'indicated_ttps' list property.

Note:
If the 'v' parameter is not an instance of :
class:`stix.common.related.RelatedTTP` an attempt will be made
to convert it to one.

Args:
v: An instance of :class:`stix.common.related.RelatedTTP` or
   :class:`stix.ttp.TTP`.

Raises:
    ValueError: If the 'v' parameter cannot be converted into an
    instance of :class:`stix.common.related.RelatedTTP`

    self.indicated_ttps.append(v)

@property
def test_mechanisms(self):
    return self._test_mechanisms

@test_mechanisms.setter
def test_mechanisms(self, value):
    self._test_mechanisms = TestMechanisms(value)

def add_test_mechanism(self, tm):
    """Adds an Test Mechanism to the 'test_mechanisms' list property
    of this 'Indicator'.

    The 'tm' parameter must be an instance of a
    :class:`stix.indicator.test_mechanism._BaseTestMechanism`
    implementation.

    If the 'tm' parameter is 'None', no item will be added to the
    'test_mechanisms' list property.

    See Also:
    Test Mechanism implementations are found under the
    :mod:`stix.extensions.test_mechanism` package.

    Args:
    tm: An instance of a
        :class:`stix.indicator.test_mechanism._BaseTestMechanism`
        implementation.

    Raises:
    ValueError: If the 'tm' parameter is not an instance of
        :class:`stix.indicator.test_mechanism._BaseTestMechanism`
        
        self.test_mechanisms.append(tm)

@property
def related_indicators(self):
    return self._related_indicators

@related_indicators.setter
def related_indicators(self, value):
    if isinstance(value, RelatedIndicators):
        self._related_indicators = value
    else:
        self._related_indicators = RelatedIndicators(value)

def add_related_indicator(self, indicator):
    """Adds an Related Indicator to the 'related_indicators' list
    property of this 'Indicator'.

    Jason Mack, jasonmack@gmail.com
The `indicator` parameter must be an instance of `stix.common.related.RelatedIndicator` or `Indicator`.

If the `indicator` parameter is `None`, no item will be added to the `related_indicators` list property.

Calling this method is the same as calling `append()` on the `related_indicators` property.

See Also:
The `RelatedIndicators` documentation.

Note:
If the `tm` parameter is not an instance of `stix.common.related.RelatedIndicator` an attempt will be made to convert it to one.

Args:
indicator: An instance of `stix.common.related.RelatedIndicator`.

Raises:
ValueError: If the `indicator` parameter cannot be converted into an instance of `stix.common.related.RelatedIndicator`.

```python
self.related_indicators.append(indicator)
```

@property
def related_campaigns(self):
    return self._related_campaigns

@related_campaigns.setter
def related_campaigns(self, value):
    if isinstance(value, RelatedCampaignRefs):
        self._related_campaigns = value
    else:
        self._related_campaigns = RelatedCampaignRefs(value)

def add_related_campaign(self, value):
    """Adds a Related Campaign to this Indicator.
    The `value` parameter must be an instance of `RelatedCampaignRef` or `CampaignRef`.
    If the `value` parameter is `None`, no item will be added to the `related_campaigns` collection.
    Calling this method is the same as calling `append()` on the `related_campaigns` property.
    See Also:
The `RelatedCampaignRef` documentation.
    Note:
    If the `value` parameter is not an instance of `RelatedCampaignRef` an attempt will be made to convert it to one.
    Args:
    value: An instance of `RelatedCampaignRef` or `Campaign`.
    Raises:
    ValueError: If the `value` parameter cannot be converted into an instance of `RelatedCampaignRef`.
```
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
self.related_campaigns.append(value)

@property
def observable_composition_operator(self):
    return self._observable_composition_operator

@observable_composition_operator.setter
def observable_composition_operator(self, value):
    if value in self._ALLOWED_COMPOSITION_OPERATORS:
        self._observable_composition_operator = value
    return

error = "observable_composition_operator must one of {0}"" = error.format(self._ALLOWED_COMPOSITION_OPERATORS)
raise ValueError(error)

@property
def likely_impact(self):
    return self._likely_impact

@likely_impact.setter
def likely_impact(self, value):
    self._set_var(Statement, likely_impact=value)

@property
def negate(self):
    return self._negate

@negate.setter
def negate(self, value):
    self._negate = utils.xml_bool(value)

@property
def kill_chain_phases(self):
    return self._kill_chain_phases

@kill_chain_phases.setter
def kill_chain_phases(self, value):
    self._kill_chain_phases = KillChainPhasesReference(value)

def add_kill_chain_phase(self, value):
    """Add a new Kill Chain Phase reference to this Indicator.

    Args:
    value: a :class:`stix.common.kill_chains.KillChainPhase` or a `str`
    representing the phase_id of. Note that you if you are defining
    a custom Kill Chain, you need to add it to the STIX package
    separately.
    """
    self._kill_chain_phases.append(value)

@property
def related_packages(self):
    return self._related_packages

@related_packages.setter
def related_packages(self, value):
    self._related_packages = RelatedPackageRefs(value)

def add_related_package(self, value):
    """Sets the name of the producer of this indicator.

    This is the same as calling
    """"indicator.producer.identity.name = identity""
    """
```

Jason Mack, jasonmack@gmail.com
If the `producer` property is `None`, it will be initialized to an instance of
``stix.common.information_source.InformationSource``.

If the `identity` property of the `producer` instance is `None`,
it will be initialized to an instance of
``stix.common.identity.Identity``.

Note:
if the `identity` parameter is not an instance
``stix.common.identity.Identity`` an attempt will be made
to convert it to one.

Args:
identity: An instance of `str` or
``stix.common.identity.Identity``.

```python
def unset_producer_identity():
    try:
        self.producer.identity.name = None
    except AttributeError:
        pass

if not identity:
    unset_producer_identity()
    return

if not self.producer:
    self.producer = InformationSource()

if isinstance(identity, Identity):
    self.producer.identity = identity
    return

if not self.producer.identity:
    self.producer.identity = Identity()

    self.producer.identity.name = str(identity)
```

```python
def set_produced_time(self, produced_time):
    """Sets the `produced_time` property of the `producer` property
instance fo `produced_time`.
This is the same as calling
``indicator.producer.time.produced_time = produced_time``.

The `produced_time` parameter must be an instance of `str`,
``datetime.datetime``, or `cybox.common.DateTimeWithPrecision``.

Note:
If `produced_time` is a `str` or `datetime.datetime` instance
an attempt will be made to convert it into an instance of
``cybox.common.DateTimeWithPrecision``.

Args:
produced_time: An instance of `str`,
``datetime.datetime``, or `cybox.common.DateTimeWithPrecision``.

```python
```
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
def get_produced_time(self):
    """Gets the produced time for this :class:`Indicator`.
    This is the same as calling
    `produced_time = indicator.producer.time.produced_time`.
    Returns:
    `None` or an instance of `cybox.common.DateTimeWithPrecision`.
    """
    try:
        return self.producer.time.produced_time
    except AttributeError:
        return None

def set_received_time(self, received_time):
    """Sets the received time for this :class:`Indicator`.
    This is the same as calling
    `indicator.producer.time.produced_time = produced_time`.
    The `received_time` parameter must be an instance of `str`,
    `datetime.datetime`, or `cybox.common.DateTimeWithPrecision`.
    Args:
    received_time: An instance of `str`,
                   `datetime.datetime`, or `cybox.common.DateTimeWithPrecision`.
    Note:
    If `received_time` is a `str` or `datetime.datetime` instance
    an attempt will be made to convert it into an instance of
    `cybox.common.DateTimeWithPrecision`.
    """
    if not self.producer:
        self.producer = InformationSource()
    if not self.producer.time:
        self.producer.time = Time()
    self.producer.time.received_time = received_time

def get_received_time(self):
    """Gets the received time for this :class:`Indicator`.
    This is the same as calling
    `received_time = indicator.producer.time.received_time`.
    Returns:
    `None` or an instance of `cybox.common.DateTimeWithPrecision`.
    """
    try:
        return self.producer.time.received_time
    except AttributeError:
        return None

def _merge_observables(self, observables):
    observable_composition = ObservableComposition()
    observable_composition.operator = self.observable_composition_operator
    for observable in observables:
        observable_composition.add(observable)
    root_observable = Observable()
    root_observable.observable_composition = observable_composition
    return root_observable

def add_object(self, object_):
```

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

"""Adds a python-cybox Object instance to the `observables` list property.

This is the same as calling `indicator.add_observable(object_)`.

Note:
If the `object` param is not an instance of `cybox.core.Object` an attempt will be made to convert it into one before wrapping it in an `cybox.core.Observable` layer.

Args:
  object_: An instance of `cybox.core.Object` or an object that can be converted into an instance of `cybox.core.Observable`

Raises:
  ValueError: if the `object_` param cannot be converted to an instance of `cybox.core.Observable`.

```python
if not object_
    return

observable = Observable(object_)
self.add_observable(observable)
```

def to_obj(self, return_obj=None, ns_info=None):
    if not return_obj:
        return_obj = self._binding_class()
    super(Indicator, self).to_obj(return_obj=return_obj, ns_info=ns_info)
    return_obj.negate = True if self.negate else None
    if self.confidence:
        return_obj.Confidence = self.confidence.to_obj(ns_info=ns_info)
    if self.indicator_types:
        return_obj.Type = self.indicator_types.to_obj(ns_info=ns_info)
    if self.indicated_ttps:
        return_obj.Indicated_TTP = self.indicated_ttps.to_obj(ns_info=ns_info)
    if self.producer:
        return_obj.Producer = self.producer.to_obj(ns_info=ns_info)
    if self.test_mechanisms:
        return_obj.Test_Mechanisms = self.test_mechanisms.to_obj(ns_info=ns_info)
    if self.likely_impact:
        return_obj.Likely_Impact = self.likely_impact.to_obj(ns_info=ns_info)
    if self.alternative_id:
        return_obj.Alternative_ID = self.alternative_id
    if self.valid_time_positions:
        return_obj.Valid_Time_Position = self.valid_time_positions.to_obj(ns_info=ns_info)
    if self.suggested_coas:
        return_obj.Suggested_COAs = self.suggested_coas.to_obj(ns_info=ns_info)
    if self.sightings:
        return_obj.Sightings = self.sightings.to_obj(ns_info=ns_info)
    if self.composite_indicator_expression:
        return_obj.Composite_Indicator_Expression = self.composite_indicator_expression.to_obj(ns_info=ns_info)
    if self.kill_chain_phases:
        return_obj.Kill_Chain_Phases = self.kill_chain_phases.to_obj(ns_info=ns_info)
    if self.related_indicators:
        return_obj.Related_Indicators = self.related_indicators.to_obj(ns_info=ns_info)
    if self.related_campaigns:
        return_obj.Related_Campaigns = self.related_campaigns.to_obj(ns_info=ns_info)
    if self.related_packages:
        return_obj.Related_Packages = self.related_packages.to_obj(ns_info=ns_info)
    if self.observables:
        if len(self.observables) > 1:
            root_observable = self._merge_observables(self.observables)
        else:
            root_observable = self.observables[0]
        return_obj.Observable = root_observable.to_obj(ns_info=ns_info)
```
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

return_obj.indicator_types = IndicatorTypes.from_list(get('indicator_types'))
return_obj.confidence = Confidence.from_dict(get('confidence'))
return_obj.valid_time_positions = _ValidTimePositions.from_dict(get('valid_time_positions'))
return_obj.observable = Observable.from_dict(get('observable'))
return_obj.producer = InformationSource.from_dict(get('producer'))
return_obj.related_campaigns = RelatedCampaignRefs.from_dict(get('related_campaigns'))
return_obj.related_packages = RelatedPackageRefs.from_dict(get('related_packages'))

return return_obj

class CompositeIndicatorExpression(stix.EntityList):
    """Implementation of the STIX CompositeIndicatorExpressionType."

The CompositeIndicatorExpression class implements methods found on collections.MutableSequence and as such can be interacted with as a list (e.g., append()).

Note:
The append() method can only accept instances of class: Indicator.

Examples:
Add a class: Indicator instance to an instance of class: CompositeIndicatorExpression:

>>> i = Indicator()
>>> comp = CompositeIndicatorExpression()
>>> comp.append(i)

Create a class: CompositeIndicatorExpression from a list of class: Indicator instances using *args argument list:

>>> list_indicators = [Indicator() for i in xrange(10)]
>>> comp = CompositeIndicatorExpression(CompositeIndicatorExpression.OP_OR, *list_indicators)
>>> len(comp)
10

Args:
operator (str, optional): The logical composition operator. Must be "AND" or "OR".
*args: Variable length argument list of class: Indicator instances.

Attributes:
OP_AND (str): String "AND"
OP_OR (str): String "OR"
OPERATORS (tuple): Tuple of allowed operator values.
operator (str): The logical composition operator. Must be "AND" or "OR".

===
._binding = indicator_binding
_binding_class = indicator_binding.CompositeIndicatorExpressionType
_namespace = "http://stix.mitre.org/Indicator-2"
_containered_type = Indicator
_binding_var = "Indicator"
_inner_name = "indicators"
OP_AND = "AND"
OP_OR = "OR"
OPERATORS = (OP_AND, OP_OR)

def __init__(self, operator="OR", *args):
    super(CompositeIndicatorExpression, self).__init__(*args)
    self.operator = operator

@property
def operator(self):
    return self._operator

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

@operator.setter
def operator(self, value):
    if not value:
        raise ValueError("operator must not be None or empty")
    elif value not in self.OPERATORS:
        raise ValueError("operator must be one of: %s" % (self.OPERATORS,))
    else:
        self._operator = value

def __nonzero__(self):
    return super(CompositeIndicatorExpression, self).__nonzero__()

def to_obj(self, return_obj=None, ns_info=None):
    list_obj = super(CompositeIndicatorExpression, self).to_obj(return_obj=return_obj, ns_info=ns_info)
    list_obj.operator = self.operator
    return list_obj

def to_dict(self):
    d = super(CompositeIndicatorExpression, self).to_dict()
    if self.operator:
        d['operator'] = self.operator
    return d

@classmethod
def from_obj(cls, obj, return_obj=None):
    if not obj:
        return None
    if return_obj is None:
        return_obj = cls()
    super(CompositeIndicatorExpression, cls).from_obj(obj, return_obj=return_obj)
    return_obj.operator = obj.operator
    return return_obj

@classmethod
def from_dict(cls, dict_repr, return_obj=None):
    if not dict_repr:
        return None
    if return_obj is None:
        return_obj = cls()
    super(CompositeIndicatorExpression, cls).from_dict(dict_repr, return_obj=return_obj)
    return_obj.operator = dict_repr.get('operator')
    return return_obj

class RelatedCampaignRefs(GenericRelationshipList):
    _namespace = "http://stix.mitre.org/Indicator-2"
    _binding = indicator_binding
    _binding_class = _binding.RelatedCampaignReferencesType
    _binding_var = 'Related_Campaign'
    _contained_type = RelatedCampaignRef
    _inner_name = "related_campaigns"

    def __init__(self, related_campaign_refs=None, scope=None):
        super(RelatedCampaignRefs, self).__init__(scope, related_campaign_refs)

    def _fix_value(self, value):
        from stix.campaign import Campaign
        if isinstance(value, Campaign) and value.id_:
            return RelatedCampaignRef(CampaignRef(idref=value.id_))
        else:
            return super(RelatedCampaignRefs, self)._fix_value(value)

# NOT ACTUAL STIX TYPES!
class IndicatorTypes(stix.TypedList):
    """A :class:`stix.common.vocabs.VocabString` collection which defaults to
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

:class:`stix.common.vocabs.IndicatorType`. This class implements methods found on `collections.MutableSequence` and as such can be interacted with like a `list`.

Note:
The `append()` method can accept `str` or :class:`stix.common.vocabs.VocabString` instances. If a `str` instance is passed in, an attempt will be made to convert it to an instance of :class:`stix.common.vocabs.IndicatorType`.

Examples:
Add an instance of :class:`stix.common.vocabs.IndicatorType`:

```python
>>> from stix.common.vocabs import IndicatorType
>>> itypes = IndicatorTypes()
>>> type = IndicatorType(IndicatorType.TERM_IP_WATCHLIST)
>>> itypes.append(type)
>>> print len(itypes)
1
```

Add a string value:

```python
>>> from stix.common.vocabs import IndicatorType
>>> itypes = IndicatorTypes()
>>> type(IndicatorType.TERM_IP_WATCHLIST)
<type 'str'>
>>> itypes.append(IndicatorType.TERM_IP_WATCHLIST)
>>> print len(itypes)
1
```

Args:
*args: Variable length argument list of strings or :class:`stix.common.vocabs.VocabString` instances.

```python
==
_namespace = "http://stix.mitre.org/Indicator-2"
_contained_type = VocabString

def _fix_value(self, value):
    return IndicatorType(value)

class _IndicatedTTPs(stix.TypedList):
    _contained_type = RelatedTTP

class _Observables(stix.TypedList):
    _contained_type = Observable
```

8.3. Python Source Code for LibTaxii – Base Code for TAXII Clients

The following scripts is by MITRE Corporation which is free to use according to their license. It has been included here so that the reader can readily study the code to better understand the technical details of TAXII.

# Copyright (C) 2013 - The MITRE Corporation
# For license information, see the LICENSE.txt file

# Contributors:
# * Alex Ciobanu - calex@cert.europa.eu
# * Mark Davidson - mdavidson@mitre.org
# * Bryan Worrell - bworrell@mitre.org
# * Benjamin Yates - byates@dtcc.com

Jason Mack, jasonmack@gmail.com
Creating, handling, and parsing TAXII 1.0 messages.

```python
try:
    import simplejson as json
except ImportError:
    import json
import os
import StringIO
import warnings
from lxml import etree
from .common import (parse, parse_datetime_string, append_any_content_etree,
                     TAXIIBase,
                     get_required, get_optional, get_optional_text)
from .validation import do_check, uri_regex, check_timestamp_label, message_id_regex_10
from constants import *

def validate_xml(xml_string):
    warnings.warn('Call to deprecated function: libtaxii.messages_10.validate_xml()',
                  category=DeprecationWarning)
    if isinstance(xml_string, basestring):
        f = StringIO.StringIO(xml_string)
    else:
        f = xml_string
    etree_xml = parse(f)
    package_dir, package_filename = os.path.split(__file__)
    schema_file = os.path.join(package_dir, "xsd", "TAXII_XMLMessageBinding_Schema.xsd")
    taxii_schema_doc = parse(schema_file)
    xml_schema = etree.XMLSchema(taxii_schema_doc)
    valid = xml_schema.validate(etree_xml)
    if not valid:
        return xml_schema.error_log.last_error
    return valid

def get_message_from_xml(xml_string):
    is_valid = tm10.validate_xml(message.to_xml())
    warnings.warn('Call to deprecated function: libtaxii.messages_10.validate_xml()',
                  category=DeprecationWarning)
    if isinstance(xml_string, basestring):
        f = StringIO.StringIO(xml_string)
    else:
        f = xml_string
    etree_xml = parse(f)
    package_dir, package_filename = os.path.split(__file__)
    schema_file = os.path.join(package_dir, "xsd", "TAXII_XMLMessageBinding_Schema.xsd")
    taxii_schema_doc = parse(schema_file)
    xml_schema = etree.XMLSchema(taxii_schema_doc)
    valid = xml_schema.validate(etree_xml)
    if not valid:
        return xml_schema.error_log.last_error
    return valid
```

This function automatically detects which type of Message should be created based on the XML.

Args:
    xml_string (str): The XML to parse into a TAXII message.

Example:
```python
message_xml = message.to_xml()
new_message = tm10.get_message_from_xml(message_xml)
```
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
***
if isinstance(xml_string, basestring):
    f = StringIO.StringIO(xml_string)
else:
    f = xml_string

etree_xml = parse(f)
qn = etree.QName(etree_xml)
if qn.namespace != ns_map['taxii']:
    raise ValueError('Unsupported namespace: %s' % qn.namespace)

message_type = qn.localname

if message_type == MSG_DISCOVERY_REQUEST:
    return DiscoveryRequest.from_etree(etree_xml)
if message_type == MSG_DISCOVERY_RESPONSE:
    return DiscoveryResponse.from_etree(etree_xml)
if message_type == MSG_FEED_INFORMATION_REQUEST:
    return FeedInformationRequest.from_etree(etree_xml)
if message_type == MSG_FEED_INFORMATION_RESPONSE:
    return FeedInformationResponse.from_etree(etree_xml)
if message_type == MSG_POLL_REQUEST:
    return PollRequest.from_etree(etree_xml)
if message_type == MSG_POLL_RESPONSE:
    return PollResponse.from_etree(etree_xml)
if message_type == MSG_STATUS_MESSAGE:
    return StatusMessage.from_etree(etree_xml)
if message_type == MSG_INBOX_MESSAGE:
    return InboxMessage.from_etree(etree_xml)
if message_type == MSG_MANAGE_FEED_SUBSCRIPTION_REQUEST:
    return ManageFeedSubscriptionRequest.from_etree(etree_xml)
if message_type == MSG_MANAGE_FEED_SUBSCRIPTION_RESPONSE:
    raise ValueError('Unsupported namespace: %s' % qn.namespace)

raise ValueError('Unknown message_type: %s' % message_type)

def get_message_from_dict(d):
    ***Create a TAXII Message object from a dictionary.
    This function automatically detects which type of Message should be created
    based on the 'message_type' key in the dictionary.

    Args:
    d (dict): The dictionary to build the TAXII message from.

    Example:
    .. code-block:: python

        message_dict = message.to_dict()
        new_message = tm10.get_message_from_dict(message_dict)

    ***
    if 'message_type' not in d:
        raise ValueError('message_type is a required field!')
    message_type = d['message_type']
    if message_type == MSG_DISCOVERY_REQUEST:
        return DiscoveryRequest.from_dict(d)
    if message_type == MSG_DISCOVERY_RESPONSE:
        return DiscoveryResponse.from_dict(d)
    if message_type == MSG_FEED_INFORMATION_REQUEST:
        return FeedInformationRequest.from_dict(d)
    if message_type == MSG_FEED_INFORMATION_RESPONSE:
        return FeedInformationResponse.from_dict(d)
    if message_type == MSG_POLL_REQUEST:
        return PollRequest.from_dict(d)
    if message_type == MSG_POLL_RESPONSE:
        return PollResponse.from_dict(d)
    if message_type == MSG_STATUS_MESSAGE:
        return StatusMessage.from_dict(d)
```

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com
def inbox_address(self):
    return self._inbox_address

@inbox_address.setter
def inbox_address(self, value):
    # TODO: Can inbox_address be validated?
    self._inbox_address = value

@property
def delivery_message_binding(self):
    return self._delivery_message_binding

@delivery_message_binding.setter
def delivery_message_binding(self, value):
    do_check(value, 'delivery_message_binding', regex_tuple=uri_regex)
    self._delivery_message_binding = value

@property
def content_bindings(self):
    return self._content_bindings

@content_bindings.setter
def content_bindings(self, value):
    do_check(value, 'content_bindings', regex_tuple=uri_regex)
    self._content_bindings = value

def to_etree(self):
    xml = etree.Element('{%s}Push_Parameters' % ns_map['taxii'])
    if self.inbox_protocol is not None:
        pb = etree.SubElement(xml, '{%s}Protocol_Binding' % ns_map['taxii'])
        pb.text = self.inbox_protocol
    if self.inbox_address is not None:
        a = etree.SubElement(xml, '{%s}Address' % ns_map['taxii'])
        a.text = self.inbox_address
    if self.delivery_message_binding is not None:
        mb = etree.SubElement(xml, '{%s}Message_Binding' % ns_map['taxii'])
        mb.text = self.delivery_message_binding
    for binding in self.content_bindings:
        cb = etree.SubElement(xml, '{%s}Content_Binding' % ns_map['taxii'])
        cb.text = binding
    return xml

def to_dict(self):
    d = {}
    if self.inbox_protocol is not None:
        d['inbox_protocol'] = self.inbox_protocol
    if self.inbox_address is not None:
        d['inbox_address'] = self.inbox_address
    if self.delivery_message_binding is not None:
        d['delivery_message_binding'] = self.delivery_message_binding
    d['content_bindings'] = []
    for binding in self.content_bindings:
        d['content_bindings'].append(binding)
    return d

def to_text(self, line_prepend=''):  
    s = line_prepend + '=== Push Parameters ===
    s += line_prepend + '  Inbox Protocol: %s
    s += line_prepend + '  Address: %s

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

s += line_prepend + " Message Binding: %s\n" % self.delivery_message_binding
if len(self.content_bindings) > 0:
    s += line_prepend + " Content Bindings: Any Content\n"
for cb in self.content_bindings:
    s += line_prepend + " Content Binding: %s\n" % str(cb)
return s

@classmethod
def from_etree(etree_xml):
    inbox_protocol = get_optional_text(etree_xml, './taxii:Protocol_Binding', ns_map)
    inbox_address = get_optional_text(etree_xml, './taxii:Address', ns_map)
    delivery_message_binding = get_optional_text(etree_xml, './taxii:Message_Binding', ns_map)
    content_bindings = []
    for binding in etree_xml.xpath('./taxii:Content_Binding', namespaces=ns_map):
        content_bindings.append(binding.text)
    return DeliveryParameters(inbox_protocol, inbox_address, delivery_message_binding, contentBindings)

@classmethod
def from_dict(d):
    return DeliveryParameters(**d)

class TAXIIIMessage(TAXIIBase):
    """Encapsulate properties common to all TAXII Messages (such as headers).

    This class is extended by each Message Type (e.g., DiscoveryRequest), with
    each subclass containing subclass-specific information
    """
    message_type = 'TAXIIIMessage'
    
def __init__(self, message_id=None, in_response_to=None, extended_headers=None):
        """Create a new TAXIIIMessage

        Arguments:
        - message_id (string) - A value identifying this message.
        - in_response_to (string) - Contains the Message ID of the message to
          which this is a response.
        - extended_headers (dictionary) - A dictionary of name/value pairs for
          use as Extended Headers
        """
        self.message_id = message_id
        self.in_response_to = in_response_to
        if extended_headers is None:
            self.extended_headers = {}
        else:
            self.extended_headers = extended_headers
        @property
def message_id(self):
            return self._message_id
        @message_id.setter
def message_id(self, value):
            do_check(value, 'message_id', regex_tuple=message_id_regex_10)
            self._message_id = value
        @property
def in_response_to(self):
            return self._in_response_to
        @in_response_to.setter
def in_response_to(self, value):
            do_check(value, 'in_response_to', regex_tuple=message_id_regex_10, can_be_none=True)
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

s += line_prepend + "Message ID: %s" % self.message_id
if self.in_response_to:
    s += "; In Response To: %s" % self.in_response_to
s += "\n"
for k, v in self.extended_headers.iteritems():
    s += line_prepend + "Extended Header: %s = %s" % (k, v)
return s

@classmethod
def from_etree(cls, src_etree, **kwargs):
    """Pulls properties of a TAXII Message from an etree.

    Message-specific constructs must be pulled by each Message class. In general, when converting from etree, subclasses should call this method first, then parse their specific XML constructs.
    """
    
    # Check namespace and element name of the root element
    expected_tag = '{%s}%s' % (ns_map['taxii'], cls.message_type)
tag = src_etree.tag
if tag != expected_tag:
    raise ValueError('%s != %s' % (tag, expected_tag))

    # Get the message ID
    message_id = get_required(src_etree, '/taxii:*/@message_id', ns_map)

    # Get in response to, if present
    in_response_to = get_optional(src_etree, '/taxii:*/@in_response_to', ns_map)
if in_response_to:
    kwargs['in_response_to'] = in_response_to

    # Get the Extended headers
    extended_header_list = src_etree.xpath('/taxii:*/taxii:Extended_Headers/taxii:Extended_Header', namespaces=ns_map)
    extended_headers = {}
    for header in extended_header_list:
        eh_name = header.xpath('./@name')[0]
        # eh_value = header.text
        if len(header) == 0:  # This has string content
            eh_value = header.text
        else:  # This has XML content
            eh_value = header[0]
        extended_headers[eh_name] = eh_value
    return cls(message_id, extended_headers=extended_headers, **kwargs)

@classmethod
def from_xml(cls, xml):
    """Parse a Message from XML.

    Subclasses shouldn't implement this method, as it is mainly a wrapper for cls.from_etree.
    """
    if isinstance(xml, basestring):
f = StringIO.StringIO(xml)
else:
f = xml
etree_xml = parse(f)
return cls.from_etree(etree_xml)

@classmethod
def from_dict(cls, d, **kwargs):
    """Pulls properties of a TAXII Message from a dictionary.

    Message-specific constructs must be pulled by each Message class. In general, when converting from dictionary, subclasses should call this
    """

Jason Mack, jasonmack@gmail.com
method first, then parse their specific dictionary constructs.

```python
message_type = d["message_type"]
if message_type != cls.message_type:
    raise ValueError(\'%s != %s\' % (message_type, cls.message_type))
message_id = d["message_id"]
extended_headers = {}
for k, v in d["extended_headers"].iteritems():
    try:
        v = parse(v)
    except etree.XMLSyntaxError:
        pass
    extended_headers[k] = v
in_response_to = d.get(\'in_response_to\')
if in_response_to:
    kwargs[\'in_response_to\'] = in_response_to
return cls(message_id, extended_headers=extended_headers, **kwargs)
```

```python
@classmethod
def from_json(cls, json_string):
    return cls.from_dict(json.loads(json_string))
```

```python
class ContentBlock(TAXIIBase):
    """A TAXII Content Block."
    Args:
        content_binding (str): a Content Binding ID or nesting expression
            indicating the type of content contained in the Content field of this
            Content Block. **Required**
        content (string or etree): a piece of content of the type specified
            by the Content Binding. **Required**
        timestamp_label (datetime): the Timestamp Label associated with this
            Content Block. **Optional**
        padding (string): an arbitrary amount of padding for this Content
            Block. **Optional**
    """
    NAME = 'Content_Block'

def __init__(self, content_binding, content, timestamp_label=None, padding=None):
    self.content_binding = content_binding
    self.content, self.content_is_xml = self._stringify_content(content)
    self.timestamp_label = timestamp_label
    self.padding = padding

@property
def sort_key(self):
    return self.content[:25]

@property
def content_binding(self):
    return self._content_binding

@content_binding.setter
def content_binding(self, value):
    do_check(value, 'content_binding', regex_tuple=uri_regex)
    self._content_binding = value

@property
def content(self):
    if self.content_is_xml:
        return etree.tostring(self._content)
    else:
        return self._content
```

```python
Jason Mack, jasonmack@gmail.com
```
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

@content.setter
def content(self, value):
    do_check(value, 'content') # Just check for not None
    self._content, self.content_is_xml = self._stringify_content(value)

@property
def content_is_xml(self):
    return self._content_is_xml

@property
def content_is_xml.setter
def content_is_xml(self, value):
    do_check(value, 'content_is_xml', value_tuple=(True, False))
    self._content_is_xml = value

def _stringify_content(self, content):
    """Always a string or raises an error.
    Returns the string representation and whether the data is XML.
    ""
    if isinstance(content, etree._ElementTree):
        return content.getroot(), True
    if isinstance(content, etree._Element):
        return content, True
    if hasattr(content, 'read'): # The content is file-like
        try:
            xml = parse(content)
        except etree.XMLSyntaxError:
            return content.read(), False
        else:
            try:
                xml = parse(StringIO.StringIO(content))
            except etree.XMLSyntaxError:
                return content.read(), False
            else:
                if isinstance(content, basestring): # It's a string of some kind, unicode or otherwise
                    return content, False
                else:
                    raise
    else:
        if isinstance(content, etree._ElementTree):
            return content.getroot(), True
        if issubclass(type(content), basestring):
            return content, False
        else:
            raise

def to_etree(self):
    block = etree.Element('{%s}Content_Block' % ns_map['taxii'], nsmap=ns_map)
    cb = etree.SubElement(block, '{%s}Content_Binding' % ns_map['taxii'])
    cb.text = self.content_binding
    c = etree.SubElement(block, '{%s}Content' % ns_map['taxii'])
    if self.content_is_xml:
        c.append(self._content)
    else:
        c.text = self._content
    if self.timestamp_label:
        tl = etree.SubElement(block, '{%s}Timestamp_Label' % ns_map['taxii'])
        tl.text = self.timestamp_label.isoformat()
    if self.padding is not None:
        p = etree.SubElement(block, '{%s}Padding' % ns_map['taxii'])
        p.text = self.padding
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
def to_dict(self):
    block = {}
    block['content_binding'] = self.content_binding
    if self.content_is_xml:
        block['content'] = etree.tostring(self._content)
    else:
        block['content'] = self._content
    block['content_is_xml'] = self.content_is_xml
    if self.timestamp_label:
        block['timestamp_label'] = self.timestamp_label.isoformat()
    if self.padding is not None:
        block['padding'] = self.padding
    return block

def to_json(self):
    return json.dumps(self.to_dict())

def to_text(self, line_prepend):
    s = line_prepend + '===' Content Block ===
    s += line_prepend + '  Content Binding: %s
    s += line_prepend + '  Content Length: %s
    s += line_prepend + ' (Only content length is shown for brevity)
    if self.timestamp_label:
        s += line_prepend + '  Timestamp Label: %s
    s += line_prepend + '  Padding: %s
    return s

@staticmethod
def from_etree(etree_xml):
    kwargs = {}
    kwargs['content_binding'] = get_required(etree_xml, './taxii:Content_Binding', ns_map).text
    kwargs['padding'] = get_optional_text(etree_xml, './taxii:Padding', ns_map)
    ts_text = get_optional_text(etree_xml, './taxii:Timestamp_Label', ns_map)
    if ts_text:
        kwargs['timestamp_label'] = parse_datetime_string(ts_text)
    content = get_required(etree_xml, './taxii:Content', ns_map)
    if len(content) == 0:  # This has string content
        kwargs['content'] = content.text
    else:  # This has XML content
        kwargs['content'] = content[0]
    return ContentBlock(**kwargs)

@staticmethod
def from_dict(d):
    kwargs = {}
    kwargs['content_binding'] = d['content_binding']
    kwargs['padding'] = d.get('padding')
    if d.get('timestamp_label'):
        kwargs['timestamp_label'] = parse_datetime_string(d['timestamp_label'])
    is_xml = d.get('content_is_xml', False)
    if is_xml:
        #FIXME: to parse or not to parse the content - this should be configurable
```

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

kwargs['content'] = parse(d['content'])
else:
    kwargs['content'] = d['content']

cb = ContentBlock(**kwargs)
return cb

@classmethod
def from_json(cls, json_string):
    return cls.from_dict(json.loads(json_string))

# TAXII Message Classes #
class DiscoveryRequest(TAXIIMessage):

    """
    A TAXII Discovery Request message.
    """

    Args:
        message_id (str): A value identifying this message. **Required**
        extended_headers (dict): A dictionary of name/value pairs for
            use as Extended Headers. **Optional**

    message_type = MSG_DISCOVERY_REQUEST

    @TAXIIMessage.in_response_to.setter
def in_response_to(self, value):
        if value:
            raise ValueError('in_response_to must be None')
        self._in_response_to = value

class DiscoveryResponse(TAXIIMessage):

    """
    A TAXII Discovery Response message.
    """

    Args:
        message_id (str): A value identifying this message. **Required**
        in_response_to (str): Contains the Message ID of the message to
            which this is a response. **Optional**
        extended_headers (dict): A dictionary of name/value pairs for
            use as Extended Headers. **Optional**
        service_instances (list of `ServiceInstance`): a list of
            service instances that this response contains. **Optional**

    message_type = MSG_DISCOVERY_RESPONSE

    def __init__(self, message_id, in_response_to, extended_headers=None, service_instances=None):
        super(DiscoveryResponse, self).__init__(message_id, in_response_to, extended_headers)
        self.service_instances = service_instances or []

    @TAXIIMessage.in_response_to.setter
def in_response_to(self, value):
        do_check(value, 'in_response_to', regex_tuple=uri_regex)
        self._in_response_to = value

    @property
def service_instances(self):
        return self._service_instances

    @service_instances.setter
def service_instances(self, value):
        do_check(value, 'service_instances', type=ServiceInstance)
        self._service_instances = value

def to_etree(self):

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

xml = super(DiscoveryResponse, self).to_etree()
for service_instance in self.service_instances:
    xml.append(service_instance.to_etree())
return xml

def to_dict(self):
    d = super(DiscoveryResponse, self).to_dict()
    d['service_instances'] = []
    for service_instance in self.service_instances:
        d['service_instances'].append(service_instance.to_dict())
    return d

def to_text(self, line_prepend=''
            instance)
    s = super(DiscoveryResponse, self).to_text(line_prepend)
    for si in self.service_instances:
        s += si.to_text(line_prepend + STD_INDENT)
    return s

@classmethod
def from_etree(cls, etree_xml):
    msg = super(DiscoveryResponse, cls).from_etree(etree_xml)
    msg.service_instances = []
    for service_instance in etree_xml.xpath('./taxii:Service_Instance', namespaces=ns_map):
        si = ServiceInstance.from_etr
        msg.service_instances.append(si)
    return msg

@classmethod
def from_dict(cls, d):
    msg = super(DiscoveryResponse, cls).from_dict(d)
    msg.service_instances = []
    for service_instance in d['service_instances']:
        si = ServiceInstance.from_dict(service_instance)
        msg.service_instances.append(si)
    return msg

class ServiceInstance(TAXIIBase10):
    
    The Service Instance component of a TAXII Discovery Response Message.

    Args:
    - service_type (string): identifies the Service Type of this Service Instance. **Required**
    - services_version (string): identifies the TAXII Services Specification to which this Service conforms. **Required**
    - protocol_binding (string): identifies the protocol binding supported by this Service. **Required**
    - service_address (string): identifies the network address of the TAXII Daemon that hosts this Service. **Required**
    - message_bindings (list of strings): identifies the message bindings supported by this Service instance. **Required**
    - inbox_service_accepted_content (list of strings): identifies content bindings that this Inbox Service is willing to accept. **Optional**
    - available (boolean): indicates whether the identity of the requester (authenticated or otherwise) is allowed to access this TAXII Service. **Optional**
    - message (string): contains a message regarding this Service instance. **Optional**

    The `message_bindings` list must contain at least one value.

Jason Mack, jasonmack@gmail.com
def __init__(self, service_type, services_version, protocol_binding, 
    service_address, message_bindings, 
    inbox_service_accepted_content=None, available=None, 
    message=None):
    self.service_type = service_type
    self.services_version = services_version
    self.protocol_binding = protocol_binding
    self.service_address = service_address
    self.message_bindings = message_bindings
    self.inbox_service_accepted_content = inbox_service_accepted_content or []
    self.available = available
    self.message = message

@property
def sort_key(self):
    return self.service_address

@property
def service_type(self):
    return self._service_type

@service_type.setter
def service_type(self, value):
    do_check(value, 'service_type', value_tuple=SVC_TYPES)
    self._service_type = value

@property
def services_version(self):
    return self._services_version

@services_version.setter
def services_version(self, value):
    do_check(value, 'services_version', regex_tuple=uri_regex)
    self._services_version = value

@property
def protocol_binding(self):
    return self._protocol_binding

@protocol_binding.setter
def protocol_binding(self, value):
    do_check(value, 'protocol_binding', regex_tuple=uri_regex)
    self._protocol_binding = value

@property
def service_address(self):
    return self._service_address

@service_address.setter
def service_address(self, value):
    self._service_address = value

@property
def message_bindings(self):
    return self._message_bindings

@message_bindings.setter
def message_bindings(self, value):
    do_check(value, 'message_bindings', regex_tuple=uri_regex)
    self._message_bindings = value

@property
def inbox_service_accepted_content(self):
    return self._inbox_service_accepted_content

@inbox_service_accepted_content.setter
def inbox_service_accepted_content(self, value):
    do_check(value, 'inbox_service_accepted_content', regex_tuple=uri_regex)
    self._inbox_service_accepted_content = value

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
@property
def available(self):
    return self._available

@available.setter
def available(self, value):
    do_check(value, 'available', value_tuple=(True, False), can_be_none=True)
    self._available = value

def to_etree(self):
    si = etree.Element('{%s}Service_Instance' % ns_map['taxii'])
    si.attrib['service_type'] = self.service_type
    si.attrib['service_version'] = self.services_version
    if self.available:
        si.attrib['available'] = str(self.available).lower()
    protocol_binding = etree.SubElement(si, '{%s}Protocol_Binding' % ns_map['taxii'])
    protocol_binding.text = self.protocol_binding
    service_address = etree.SubElement(si, '{%s}Address' % ns_map['taxii'])
    service_address.text = self.service_address
    for mb in self.message_bindings:
        message_binding = etree.SubElement(si, '{%s}Message_Binding' % ns_map['taxii'])
        message_binding.text = mb
    for cb in self.inbox_service_accepted_content:
        content_binding = etree.SubElement(si, '{%s}Content_Binding' % ns_map['taxii'])
        content_binding.text = cb
    if self.message is not None:
        message = etree.SubElement(si, '{%s}Message' % ns_map['taxii'])
        message.text = self.message
    return si

def to_dict(self):
    d = {}  
    d['service_type'] = self.service_type
    d['services_version'] = self.services_version
    d['protocol_binding'] = self.protocol_binding
    d['service_address'] = self.service_address
    d['message_bindings'] = self.message_bindings
    d['inbox_service_accepted_content'] = self.inbox_service_accepted_content
    d['available'] = self.available
    d['message'] = self.message
    return d

def to_text(self, line_prepend=' '):
    s = line_prepend + '== Service Instance==
    s += line_prepend + ' Service Type: %s' % self.service_type
    s += line_prepend + ' Services Version: %s' % self.services_version
    s += line_prepend + ' Protocol Binding: %s' % self.protocol_binding
    s += line_prepend + ' Address: %s' % self.service_address
    for mb in self.message_bindings:
        s += line_prepend + ' Message Binding: %s' % mb
    if len(self.inbox_service_accepted_content) == 0:
        s += line_prepend + ' Inbox Service Accepts: %s' % None
    for isac in self.inbox_service_accepted_content:
        s += line_prepend + ' Inbox Service Accepts: %s' % isac
    s += line_prepend + ' Available: %s' % self.available
    s += line_prepend + ' Message: %s' % self.message
    return s

@classmethod
def from_etree(cls, etree_xml):
    # Expects a taxii:Service_Instance element
    service_type = etree_xml.attrib['service_type']
```

Jason Mack, jasonmack@gmail.com
services_version = etree_xml.attrib['service_version']
available = None
if etree_xml.attrib.get('available'):
    tmp_available = etree_xml.attrib['available']
    available = tmp_available.lower() == 'true'

protocol_binding = get_required(etree_xml, './taxii:Protocol_Binding', ns_map).text
service_address = get_required(etree_xml, './taxii:Address', ns_map).text

message_bindings = []
for mb in etree_xml.xpath('./taxii:Message_Binding', namespaces=ns_map):
    message_bindings.append(mb.text)

inbox_service_accepted_contents = []
for cb in etree_xml.xpath('./taxii:Content_Binding', namespaces=ns_map):
    inbox_service_accepted_contents.append(cb.text)

message = get_optional_text(etree_xml, './taxii:Message', ns_map)

return ServiceInstance(service_type, services_version, protocol_binding,
                       service_address, message_bindings, inbox_service_accepted_contents,
                       available, message)

@staticmethod
def from_dict(d):
    return ServiceInstance(**d)

class FeedInformationRequest(TAXIIMessage):
    
    A TAXII Feed Information Request message.
    
    Args:
    message_id (str): A value identifying this message. **Required**
    extended_headers (dict): A dictionary of name/value pairs for
    use as Extended Headers. **Optional**
    
    message_type = MSG_FEED_INFORMATION_REQUEST

@TAXIIMessage.in_response_to.setter
def in_response_to(self, value):
    if value:
        raise ValueError('in_response_to must be None')
    self._in_response_to = value

class FeedInformationResponse(TAXIIMessage):
    
    A TAXII Feed Information Response message.
    
    Args:
    message_id (str): A value identifying this message. **Required**
    in_response_to (str): Contains the Message ID of the message to
    which this is a response. **Required**
    extended_headers (dict): A dictionary of name/value pairs for
    use as Extended Headers. **Optional**
    feed_informations (list of FeedInformation): A list
    of FeedInformation objects to be contained in this response.
    **Optional**
    
    message_type = MSG_FEED_INFORMATION_RESPONSE

def __init__(self, message_id, in_response_to, extended_headers=None, feed_informations=None):
    super(FeedInformationResponse, self).__init__(message_id, in_response_to, extended_headers)
    self.feed_informations = feed_informations or []
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

push methods (list of PushMethod objects): the protocols that can be used to push content via a subscription. **Optional**
polling service instances (list of PollingServiceInstance objects): the bindings and address a Consumer can use to interact with a Poll Service instance that supports this TAXII Data Feed. **Optional**
subscription methods (list of SubscriptionMethod objects): the protocol and address of the TAXII Daemon hosting the Feed Management Service that can process subscriptions for this TAXII Data Feed. **Optional**

The absense of `push methods` indicates no push methods. The absense of `polling service instances` indicates no polling services. At least one of `push methods` and `polling service instances` must not be empty. The absense of `subscription methods` indicates no subscription services.

```python
def __init__(self, feed_name, feed_description, supported_contents,
            available=None, push_methods=None,
            polling_service_instances=None, subscription_methods=None):
    self.feed_name = feed_name
    self.available = available
    self.feed_description = feed_description
    self.supported_contents = supported_contents
    self.push_methods = push_methods or []
    self.polling_service_instances = polling_service_instances or []
    self.subscription_methods = subscription_methods or []

@property
def sort_key(self):
    return self.feed_name

@property
def feed_name(self):
    return self._feed_name

@feed_name.setter
def feed_name(self, value):
    do_check(value, 'feed_name', regex_tuple=uri_regex)
    self._feed_name = value

@property
def available(self):
    return self._available

@available.setter
def available(self, value):
    do_check(value, 'available', value_tuple=(True, False), can_be_none=True)
    self._available = value

@property
def supported_contents(self):
    return self._supported_contents

@supported_contents.setter
def supported_contents(self, value):
    do_check(value, 'supported_contents', regex_tuple=uri_regex)
    self._supported_contents = value

@property
def push_methods(self):
    return self._push_methods

@push_methods.setter
def push_methods(self, value):
    do_check(value, 'push_methods', type=PushMethod)
    self._push_methods = value

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
@property
def polling_service_instances(self):
    return self._polling_service_instances

@polling_service_instances.setter
def polling_service_instances(self, value):
    do_check(value, 'polling_service_instances', type=PollingServiceInstance)
    self._polling_service_instances = value

@property
def subscription_methods(self):
    return self._subscription_methods

@subscription_methods.setter
def subscription_methods(self, value):
    do_check(value, 'subscription_methods', type=SubscriptionMethod)
    self._subscription_methods = value

def to_etree(self):
    f = etree.Element('{%s}Feed' % ns_map['taxii'])
    f.attrib['feed_name'] = self.feed_name
    if self.available:
        f.attrib['available'] = str(self.available).lower()
    feed_description = etree.SubElement(f, '{%s}Description' % ns_map['taxii'])
    feed_description.text = self.feed_description
    for binding in self.supported_contents:
        cb = etree.SubElement(f, '{%s}Content_Binding' % ns_map['taxii'])
        cb.text = binding
    for push_method in self.push_methods:
        f.append(push_method.to_etree())
    for polling_service in self.polling_service_instances:
        f.append(polling_service.to_etree())
    for subscription_method in self.subscription_methods:
        f.append(subscription_method.to_etree())
    return f

def to_dict(self):
    d = {}
    d['feed_name'] = self.feed_name
    if self.available:
        d['available'] = self.available
        d['feed_description'] = self.feed_description
        d['supported_contents'] = self.supported_contents
        d['push_methods'] = []
    for push_method in self.push_methods:
        d['push_methods'].append(push_method.to_dict())
    d['polling_service_instances'] = []
    for polling_service in self.polling_service_instances:
        d['polling_service_instances'].append(polling_service.to_dict())
    d['subscription_methods'] = []
    for subscription_method in self.subscription_methods:
        d['subscription_methods'].append(subscription_method.to_dict())
    return d

def to_text(self, line_prepend=' '):
    s = line_prepend + '=== Data Feed ===
    s += line_prepend + ' Feed Name: %s' % self.feed_name
    if self.available:
        s += line_prepend + ' Available: %s
    s += line_prepend + ' Feed Description: %s
    for sc in self.supported_contents:
        s += line_prepend + ' Supported Content: %s
    for pm in self.push_methods:
        s += line_prepend + ' Push Method: %s
```
s += pm.to_text(line_prepend + STD_INDENT)
for ps in self.polling_service_instances:
    s += ps.to_text(line_prepend + STD_INDENT)
for sm in self.subscription_methods:
    s += sm.to_text(line_prepend + STD_INDENT)

return s

@staticmethod
def from_etree(etree_xml):
    kwargs = {}
    kwargs['feed_name'] = etree_xml.attrib['feed_name']
    kwargs['available'] = None
    if 'available' in etree_xml.attrib:
        tmp = etree_xml.attrib['available']
        kwargs['available'] = tmp.lower() == 'true'
    kwargs['feed_description'] = get_required(etree_xml, './taxii:Description', ns_map).text
    kwargs['supported_contents'] = []
    for binding_elt in etree_xml.xpath('./taxii:Content_Binding', namespaces=ns_map):
        kwargs['supported_contents'].append(binding_elt.text)
    kwargs['push_methods'] = []
    for push_method_elt in etree_xml.xpath('./taxii:Push_Method', namespaces=ns_map):
        kwargs['push_methods'].append(PushMethod.from_etree(push_method_elt))
    kwargs['polling_service_instances'] = []
    for polling_elt in etree_xml.xpath('./taxii:Polling_Service', namespaces=ns_map):
        kwargs['polling_service_instances'].append(PollingServiceInstance.from_etree(polling_elt))
    kwargs['subscription_methods'] = []
    for subscription_elt in etree_xml.xpath('./taxii:Subscription_Service', namespaces=ns_map):
        kwargs['subscription_methods'].append(SubscriptionMethod.from_etree(subscription_elt))

    return FeedInformation(**kwargs)

@staticmethod
def from_dict(d):
    kwargs = {}
    kwargs['feed_name'] = d['feed_name']
    kwargs['available'] = d.get('available')
    kwargs['feed_description'] = d['feed_description']
    kwargs['supported_contents'] = []
    for binding in d.get('supported_contents', []):
        kwargs['supported_contents'].append(binding)
    kwargs['push_methods'] = []
    for push_method in d.get('push_methods', []):
        kwargs['push_methods'].append(PushMethod.from_dict(push_method))
    kwargs['polling_service_instances'] = []
    for polling in d.get('polling_service_instances', []):
        kwargs['polling_service_instances'].append(PollingServiceInstance.from_dict(polling))
    kwargs['subscription_methods'] = []
    for subscription_method in d.get('subscription_methods', []):
        kwargs['subscription_methods'].append(SubscriptionMethod.from_dict(subscription_method))

    return FeedInformation(**kwargs)

class PushMethod(TAXIIBase10):
    
    The Push Method component of a TAXII Feed Information component.

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

Args:
push_protocol (str): a protocol binding that can be used
to push content to an Inbox Service instance. **Required**
push_message_bindings (list of str): the message bindings that
can be used to push content to an Inbox Service instance
using the protocol identified in the Push Protocol field.
**Required**

```python
def __init__(self, push_protocol, push_message_bindings):
    self.push_protocol = push_protocol
    self.push_message_bindings = push_message_bindings

@property
def sort_key(self):
    return self.push_protocol

@property
def push_protocol(self):
    return self._push_protocol

@push_protocol.setter
def push_protocol(self, value):
    do_check(value, 'push_protocol', regex_tuple=uri_regex)
    self._push_protocol = value

@property
def push_message_bindings(self):
    return self._push_message_bindings

@push_message_bindings.setter
def push_message_bindings(self, value):
    do_check(value, 'push_message_bindings', regex_tuple=uri_regex)
    self._push_message_bindings = value

def to_etree(self):
    x = etree.Element('{%s}Push_Method' % ns_map['taxii'])
    proto_bind = etree.SubElement(x, '{%s}Protocol_Binding' % ns_map['taxii'])
    proto_bind.text = self.push_protocol
    for binding in self.push_message_bindings:
        b = etree.SubElement(x, '{%s}Message_Binding' % ns_map['taxii'])
        b.text = binding
    return x

def to_dict(self):
    d = {}
    d['push_protocol'] = self.push_protocol
    d['push_message_bindings'] = []
    for binding in self.push_message_bindings:
        d['push_message_bindings'].append(binding)
    return d

def to_text(self, line_prepend=''):  
s = line_prepend + "===== Push Method =====
" + line_prepend + " Protocol Binding: %s" % self.push_protocol
    for mb in self.push_message_bindings:
        s += line_prepend + " Message Binding: %s" % mb
    return s
```

@staticmethod
def from_etree(etree_xml):
    kwargs = {}
    kwargs['push_protocol'] = get_required(etree_xml, './taxii:Protocol_Binding' % ns_map).text
    kwargs['push_message_bindings'] = []
    for message_binding in etree_xml.xpath('./taxii:Message_Binding', namespaces=ns_map):
        kwargs['push_message_bindings'].append(message_binding.text)
    return PushMethod(**kwargs)
class PollingServiceInstance(TAXIIBase10):

    """
    The Polling Service Instance component of a TAXII Feed
    Information component.
    """
    Args:
    poll_protocol (str): the protocol binding supported by
            this Poll Service instance. **Required**
    poll_address (str): the address of the TAXII Daemon
            hosting this Poll Service instance. **Required**
    poll_message_bindings (list of str): the message bindings
            supported by this Poll Service instance. **Required**
    """
    NAME = 'Polling_Service'

    def __init__(self, poll_protocol, poll_address, poll_message_bindings):
        self.poll_protocol = poll_protocol
        self.poll_address = poll_address
        self.poll_message_bindings = poll_message_bindings

    @property
    def sort_key(self):
        return self.poll_address

    @property
    def poll_protocol(self):
        return self._poll_protocol

    @poll_protocol.setter
    def poll_protocol(self, value):
        do_check(value, 'poll_protocol', regex_tuple=uri_regex)
        self._poll_protocol = value

    @property
    def poll_message_bindings(self):
        return self._poll_message_bindings

    @poll_message_bindings.setter
    def poll_message_bindings(self, value):
        do_check(value, 'poll_message_bindings', regex_tuple=uri_regex)
        self._poll_message_bindings = value

    def to_etree(self):
        x = etree.Element('{%s}Polling_Service' % ns_map['taxii'])
        proto_bind = etree.SubElement(x, '{%s}Protocol_Binding' % ns_map['taxii'])
        proto_bind.text = self.poll_protocol
        address = etree.SubElement(x, '{%s}Address' % ns_map['taxii'])
        address.text = self.poll_address
        for binding in self.poll_message_bindings:
            b = etree.SubElement(x, '{%s}Message_Binding' % ns_map['taxii'])
            b.text = binding
        return x

    def to_dict(self):
        d = {}
        d['poll_protocol'] = self.poll_protocol
        d['poll_address'] = self.poll_address
        d['poll_message_bindings'] = []
        for binding in self.poll_message_bindings:
            d['poll_message_bindings'].append(binding)
        return d

    def to_text(self, line_prepend=''): 
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise  

```python
s = line_prepend + "=== Poll Service Instance ===\n" s += line_prepend + " Protocol Binding: %s\n" % self.poll_protocol 
s += line_prepend + " Address: %s\n" % self.poll_address 
for mb in self.poll_message_bindings: 
s += line_prepend + " Message Binding: %s\n" % mb 
return s
```

```python
@classmethod 
def from_etree(cls, etree_xml): 
    protocol = get_required(etree_xml, './taxii:Protocol_Binding', ns_map).text 
    addr = get_required(etree_xml, './taxii:Address', ns_map).text 
    bindings = [] 
    for message_binding in etree_xml.xpath('./taxii:Message_Binding', namespaces=ns_map): 
        bindings.append(message_binding.text) 
    return cls(protocol, addr, bindings)
```

```python
@classmethod 
def from_dict(cls, d): 
    return cls(**d)
```

```python
class SubscriptionMethod(TAXIIBase10):
    """The Subscription Method component of a TAXII Feed Information component. 
    """
    NAME = 'Subscription_Service'
    def __init__(self, subscription_protocol, subscription_address, subscription_message_bindings): 
        self.subscription_protocol = subscription_protocol 
        self.subscription_address = subscription_address 
        self.subscription_message_bindings = subscription_message_bindings 

    @property 
    def sort_key(self): 
        return self.subscription_address 

    @property 
    def subscription_protocol(self): 
        return self._subscription_protocol 

    @subscription_protocol.setter 
    def subscription_protocol(self, value): 
        do_check(value, 'subscription_protocol', regex_tuple=uri_regex) 
        self._subscription_protocol = value 

    @property 
    def subscription_message_bindings(self): 
        return self._subscription_message_bindings 

    @subscription_message_bindings.setter 
    def subscription_message_bindings(self, value): 
        do_check(value, 'subscription_message_bindings', regex_tuple=uri_regex) 
        self._subscription_message_bindings = value
```

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

feed_name=None, exclusive_begin_timestamp_label=None,
inclusive_end_timestamp_label=None, subscription_id=None,
content_bindings=None):
super(PollRequest, self)._init_(message_id, extended_headers=extended_headers)
self.feed_name = feed_name
self.exclusive_begin_timestamp_label = exclusive_begin_timestamp_label
self.inclusive_end_timestamp_label = inclusive_end_timestamp_label
self.subscription_id = subscription_id
self.content_bindings = content_bindings or []

@TAXIIMessage.in_response_to.setter
def in_response_to(self, value):
    if value:
        raise ValueError('in_response_to must be None')
    self.in_response_to = value

@property
def feed_name(self):
    return self.feed_name

@feed_name.setter
def feed_name(self, value):
    do_check(value, 'feed_name', regex_tuple=uri_regex)
    self.feed_name = value

@property
def exclusive_begin_timestamp_label(self):
    return self.exclusive_begin_timestamp_label

@exclusive_begin_timestamp_label.setter
def exclusive_begin_timestamp_label(self, value):
    value = check_timestamp_label(value, 'exclusive_begin_timestamp_label', can_be_none=True)
    self.exclusive_begin_timestamp_label = value

@property
def inclusive_end_timestamp_label(self):
    return self.inclusive_end_timestamp_label

@inclusive_end_timestamp_label.setter
def inclusive_end_timestamp_label(self, value):
    value = check_timestamp_label(value, 'inclusive_end_timestamp_label', can_be_none=True)
    self.inclusive_end_timestamp_label = value

@property
def subscription_id(self):
    return self.subscription_id

@subscription_id.setter
def subscription_id(self, value):
    do_check(value, 'subscription_id', regex_tuple=uri_regex, can_be_none=True)
    self.subscription_id = value

@property
def content_bindings(self):
    return self.content_bindings

@content_bindings.setter
def content_bindings(self, value):
    do_check(value, 'content_bindings', regex_tuple=uri_regex)
    self.content_bindings = value

def to_etree(self):
    xml = super(PollRequest, self).to_etree()
    xml.attrib['feed_name'] = self.feed_name
    if self.subscription_id is not None:
        xml.attrib['subscription_id'] = self.subscription_id
    if self.exclusive_begin_timestamp_label:
        ebt = etree.SubElement(xml, '{%s}Exclusive_Begin_Timestamp' % ns_map['taxii'])

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

# TODO: Add TZ Info
ebt.text = self.exclusive_begin_timestamp_label.isoformat()
if self.inclusive_end_timestamp_label:
    iet = etree.SubElement(xml, '{%s}Inclusive_End_Timestamp' % ns_map['taxii'])
    iet.text = self.inclusive_end_timestamp_label.isoformat()
for binding in self.content_bindings:
    b = etree.SubElement(xml, '{%s}Content_Binding' % ns_map['taxii'])
    b.text = binding
return xml

def to_dict(self):
    d = super(PollRequest, self).to_dict()
    d['feed_name'] = self.feed_name
    if self.subscription_id is not None:
        d['subscription_id'] = self.subscription_id
    if self.exclusive_begin_timestamp_label:
        d['exclusive_begin_timestamp_label'] = self.exclusive_begin_timestamp_label.isoformat()
    if self.inclusive_end_timestamp_label:
        d['inclusive_end_timestamp_label'] = self.inclusive_end_timestamp_label.isoformat()
    d['content_bindings'] = []
    for bind in self.content_bindings:
        d['content_bindings'].append(bind)
    return d

def to_text(self, line_prepend=''):  # TODO: Add TZ Info
    s = super(PollRequest, self).to_text(line_prepend)
    s += line_prepend + '  Feed Name: %s
' % self.feed_name
    if self.subscription_id:
        s += line_prepend + '  Subscription ID: %s
' % self.subscription_id
    if self.exclusive_begin_timestamp_label:
        s += line_prepend + '  Excl. Begin Timestamp Label: %s
' % self.exclusive_begin_timestamp_label.isoformat()
    else:
        s += line_prepend + '  Excl. Begin Timestamp Label: %s
' % None
    if self.inclusive_end_timestamp_label:
        s += line_prepend + '  Incl. End Timestamp Label: %s
' % self.inclusive_end_timestamp_label.isoformat()
    else:
        s += line_prepend + '  Incl. End Timestamp Label: %s
' % None
    if len(self.content_bindings) == 0:
        s += line_prepend + '  Content Binding: Any Content
    for cb in self.content_bindings:
        s += line_prepend + '  Content Binding: %s
' % cb
    return s

classmethod
def from_etree(cls, etree_xml):
    kwargs = {}
    kwargs['feed_name'] = get_required(etree_xml, './@feed_name', ns_map)
    kwargs['subscription_id'] = get_optional(etree_xml, './@subscription_id', ns_map)
    ebt_text = get_optional_text(etree_xml, './taxii:Exclusive_Begin_Timestamp', ns_map)
    if ebt_text:
        kwargs['exclusive_begin_timestamp_label'] = parse_datetime_string(ebt_text)
    iet_text = get_optional_text(etree_xml, './taxii:Inclusive_End_Timestamp', ns_map)
    if iet_text:
        kwargs['inclusive_end_timestamp_label'] = parse_datetime_string(iet_text)
    kwargs['content_bindings'] = []
    for binding in etree_xml.xpath('./taxii:Content_Binding', namespaces=ns_map):
        kwargs['content_bindings'].append(binding.text)
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

class PollResponse(TAXIIMessage):
    
    A TAXII Poll Response message.

    Args:
    message_id (str): A value identifying this message. **Required**
    in_response_to (str): Contains the Message ID of the message to
    which this is a response. **Required**
    extended_headers (dict): A dictionary of name/value pairs for
    use as Extended Headers. **Optional**
    feed_name (str): the name of the TAXII Data Feed that was polled.
    **Required**
    inclusive_begin_timestamp_label (datetime): a Timestamp Label
    indicating the beginning of the range this response covers.
    **Optional**
    inclusive_end_timestamp_label (datetime): a Timestamp Label
    indicating the end of the range this response covers. **Optional**
    subscription_id (str): the Subscription ID for which this content
    is being provided. **Optional**
    message (str): additional information for the message recipient.
    **Optional**
    content_blocks (list of ContentBlock): piece of content
    and additional information related to the content. **Optional**

    message_type = MSG_POLL_RESPONSE

    def __init__(self, message_id, in_response_to, extended_headers=None, 
                 feed_name=None, inclusive_begin_timestamp_label=None, 
                 inclusive_end_timestamp_label=None, subscription_id=None, 
                 message=None, content_blocks=None):
        super(PollResponse, self).__init__(message_id, in_response_to, extended_headers)
        self.feed_name = feed_name
        self.inclusive_begin_timestamp_label = inclusive_begin_timestamp_label
        self.inclusive_end_timestamp_label = inclusive_end_timestamp_label
        self.subscription_id = subscription_id
        self.message = message
        self.content_blocks = content_blocks or []

    @TAXIIMessage.in_response_to.setter
    def in_response_to(self, value):
        do_check(value, 'in_response_to', regex_tuple=uri_regex)
        self._in_response_to = value

msg = super(PollRequest, cls).from_etree(etree_xml, **kwargs)
return msg

@classmethod
def from_dict(cls, d):
    kwars = {}
    kwars['feed_name'] = d['feed_name']
    kwars['subscription_id'] = d.get('subscription_id')
    kwars['exclusive_begin_timestamp_label'] = None
    if d.get('exclusive_begin_timestamp_label'):
        kwars['exclusive_begin_timestamp_label'] = parse_datetime_string(d['exclusive_begin_timestamp_label'])
    kwars['inclusive_end_timestamp_label'] = None
    if d.get('inclusive_end_timestamp_label'):
        kwars['inclusive_end_timestamp_label'] = parse_datetime_string(d['inclusive_end_timestamp_label'])
    kwars['content_bindings'] = d.get('content_bindings', [])

    msg = super(PollRequest, cls).from_dict(d, **kwars)
    return msg

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
@property
def feed_name(self):
    return self._feed_name

@feed_name.setter
def feed_name(self, value):
    do_check(value, 'feed_name', regex_tuple=uri_regex)
    self._feed_name = value

@property
def inclusive_end_timestamp_label(self):
    return self._inclusive_end_timestamp_label

@inclusive_end_timestamp_label.setter
def inclusive_end_timestamp_label(self, value):
    value = check_timestamp_label(value, 'inclusive_end_timestamp_label')
    self._inclusive_end_timestamp_label = value

@property
def inclusive_begin_timestamp_label(self):
    return self._inclusive_begin_timestamp_label

@inclusive_begin_timestamp_label.setter
def inclusive_begin_timestamp_label(self, value):
    value = check_timestamp_label(value, 'inclusive_begin_timestamp_label', can_be_none=True)
    self._inclusive_begin_timestamp_label = value

@property
def subscription_id(self):
    return self._subscription_id

@subscription_id.setter
def subscription_id(self, value):
    do_check(value, 'subscription_id', regex_tuple=uri_regex, can_be_none=True)
    self._subscription_id = value

@property
def content_blocks(self):
    return self._content_blocks

@content_blocks.setter
def content_blocks(self, value):
    do_check(value, 'content_blocks', type=ContentBlock)
    self._content_blocks = value

def to_etree(self):
    xml = super(PollResponse, self).to_etree()
    xml.attrib['feed_name'] = self.feed_name
    if self.subscription_id is not None:
        xml.attrib['subscription_id'] = self.subscription_id
    if self.message is not None:
        m = etree.SubElement(xml, '{%s}Message' % ns_map['taxii'])
        m.text = self.message
    if self.inclusive_begin_timestamp_label:
        ibt = etree.SubElement(xml, '{%s}Inclusive_Begin_Timestamp' % ns_map['taxii'])
        ibt.text = self.inclusive_begin_timestamp_label.isoformat()
    iet = etree.SubElement(xml, '{%s}Inclusive_End_Timestamp' % ns_map['taxii'])
    iet.text = self.inclusive_end_timestamp_label.isoformat()
    for block in self.content_blocks:
        xml.append(block.to_etree())
    return xml

def to_dict(self):
    d = super(PollResponse, self).to_dict()
```

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
d['feed_name'] = self.feed_name
if self.subscription_id is not None:
    d['subscription_id'] = self.subscription_id
if self.message is not None:
    d['message'] = self.message
if self.inclusive_begin_timestamp_label:
    d['inclusive_begin_timestamp_label'] = self.inclusive_begin_timestamp_label.isoformat()
if self.inclusive_end_timestamp_label:
    d['inclusive_end_timestamp_label'] = self.inclusive_end_timestamp_label.isoformat()
d['content_blocks'] = []
for block in self.content_blocks:
    d['content_blocks'].append(block.to_dict())
return d

def to_text(self, line_prepend=' '):
    s = super(PollResponse, self).to_text(line_prepend)
    s += line_prepend + '  Feed Name: %s
' % self.feed_name
    if self.subscription_id:
        s += line_prepend + '  Subscription ID: %s
' % self.subscription_id
    if self.message:
        s += line_prepend + '  Message: %s
' % self.message
    if self.inclusive_begin_timestamp_label:
        s += line_prepend + '  Incl. Begin Timestamp Label: %s
' % self.inclusive_begin_timestamp_label.isoformat()
    else:
        s += line_prepend + '  Incl. Begin Timestamp Label: %s
' % None
    s += line_prepend + '  Incl. End Timestamp Label: %s
' % self.inclusive_end_timestamp_label.isoformat()
    for cb in self.content_blocks:
        s += cb.to_text(line_prepend + STD_INDENT)
return s

@classmethod
def from_etree(cls, etree_xml):
    kwargs = {}
    kwargs['feed_name'] = get_required(etree_xml, './@feed_name', ns_map)
    kwargs['subscription_id'] = get_optional(etree_xml, './@subscription_id', ns_map)
    kwargs['message'] = get_optional_text(etree_xml, './taxii:Message', ns_map)
    ibts_text = get_optional_text(etree_xml, './taxii:Inclusive_Begin_Timestamp', ns_map)
    if ibts_text:
        kwargs['inclusive_begin_timestamp_label'] = parse_datetime_string(ibts_text)
    iets_text = get_required(etree_xml, './taxii:Inclusive_End_Timestamp', ns_map)
    kwargs['inclusive_end_timestamp_label'] = parse_datetime_string(iets_text)
    for block in etree_xml.xpath('./taxii:Content_Block', namespaces=ns_map):
        kwargs['content_blocks'].append(ContentBlock.from_etree(block))
    msg = super(PollResponse, cls).from_etree(etree_xml, **kwargs)
    return msg

@classmethod
def from_dict(cls, d):
    kwargs = {}
    kwargs['feed_name'] = d['feed_name']
    if d.get('message'):
        kwargs['message'] = d['message']
    kwargs['subscription_id'] = d['subscription_id']
    if d.get('inclusive_begin_timestamp_label') != None:
        kwargs['inclusive_begin_timestamp_label'] = parse_datetime_string(d['inclusive_begin_timestamp_label'])
```

```python
Jason Mack, jasonmack@gmail.com
```
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
kwargs['inclusive_end_timestamp_label'] = parse_datetime_string(d['inclusive_end_timestamp_label'])
for block in d['content_blocks']:
    kwargs['content_blocks'].append(ContentBlock.from_dict(block))
msg = super(PollResponse, cls).from_dict(d, **kwargs)
return msg

class StatusMessage(TAXIIMessage):
    ""
    A TAXII Status message.
    Args:
    message_id (str): A value identifying this message. **Required**
    in_response_to (str): Contains the Message ID of the message to
    which this is a response. **Required**
    extended_headers (dict): A dictionary of name/value pairs for
    use as Extended Headers. **Optional**
    status_type (str): One of the defined Status Types or a third-party
    defined Status Type. **Required**
    status_detail (str): A field for additional information about
    this status in a machine-readable format. **Optional or Prohibited**
    depending on `status_type`. See TAXII Specification for details.
    message (str): Additional information for the status. There is no
    expectation that this field be interpretable by a machine; it is
    instead targeted to a human operator. **Optional**
    ""
    message_type = MSG_STATUS_MESSAGE
    def __init__(self, message_id, in_response_to, extended_headers=None,
                 status_type=None, status_detail=None, message=None):
        super(StatusMessage, self).__init__(message_id, in_response_to, extended_headers=extended_headers)
        self.status_type = status_type
        self.status_detail = status_detail
        self.message = message

    @TAXIIMessage.in_response_to.setter
def in_response_to(self, value):
        do_check(value, 'in_response_to', regex_tuple=uri_regex)
        self._in_response_to = value

    @property
def status_type(self):
        return self._status_type

    @status_type.setter
def status_type(self, value):
        do_check(value, 'status_type')
        self._status_type = value

    # TODO: is it possible to check the status detail?
def to_etree(self):
        xml = super(StatusMessage, self).to_etree()
        xml.attrib['status_type'] = self.status_type

        if self.status_detail is not None:
            sd = etree.SubElement(xml, '{%s}Status_Detail' % ns_map['taxii'])
            sd.text = self.status_detail

        if self.message is not None:
            m = etree.SubElement(xml, '{%s}Message' % ns_map['taxii'])
            m.text = self.message

        return xml

def to_dict(self):

Jason Mack, jasonmack@gmail.com
```
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

d = super(StatusMessage, self).to_dict()
d['status_type'] = self.status_type
if self.status_detail is not None:
d['status_detail'] = self.status_detail
if self.message is not None:
d['message'] = self.message

return d
def to_text(self, line_prepend=''):  
s = super(StatusMessage, self).to_text(line_prepend)
s += line_prepend + " Status Type: %s\n" % self.status_type
if self.status_detail:
s += line_prepend + " Status Detail: %s\n" % self.status_detail
s += line_prepend + " Status Message: %s\n" % self.message

return s

@classmethod
def from_etree(cls, etree_xml):
    kwargs = dict(
        status_type = etree_xml.attrib['status_type'],
        status_detail = get_optional_text(etree_xml, './taxii:Status_Detail', ns_map),
        message = get_optional_text(etree_xml, './taxii:Message', ns_map),
    )
    msg = super(StatusMessage, cls).from_etree(etree_xml, **kwargs)
    return msg

@classmethod
def from_dict(cls, d):
    kwargs = dict(
        status_type = d['status_type'],
        status_detail = d.get('status_detail'),
        message = d.get('message'),
    )
    msg = super(StatusMessage, cls).from_dict(d, **kwargs)
    return msg

class InboxMessage(TAXIIMessage):
    ""
    A TAXII Inbox message.
    Args:
    message_id (str): A value identifying this message. **Required**
    extended_headers (dict): A dictionary of name/value pairs for use as Extended Headers. **Optional**
    message (str): prose information for the message recipient. **Optional**
    subscription_information (SubscriptionInformation): This field is only present if this message is being sent to provide content in accordance with an existing TAXII Data Feed subscription. **Optional**
    content_blocks (list of ContentBlock): Inbox content. **Optional**
    """

    message_type = MSG_INBOX_MESSAGE

def __init__(self, message_id, in_response_to=None, extended_headers=None, message=None, subscription_information=None, content_blocks=None):
    super(InboxMessage, self).__init__(message_id, extended_headers=extended_headers)
    self.subscription_information = subscription_information
    self.message = message
    self.content_blocks = content_blocks or []

@TAXIIMessage.in_response_to.setter
def in_response_to(self, value):
    if value:
        raise ValueError('in_response_to must be None')
    self._in_response_to = value

@property
def subscription_information(self):
    return self._subscription_information

@subscription_information.setter
def subscription_information(self, value):
    do_check(value, 'subscription_information', type=SubscriptionInformation, can_be_none=True)
    self._subscription_information = value

@property
def content_blocks(self):
    return self._content_blocks

@content_blocks.setter
def content_blocks(self, value):
    do_check(value, 'content_blocks', type=ContentBlock)
    self._content_blocks = value

def to_etree(self):
    xml = super(InboxMessage, self).to_etree()
    if self.message is not None:
        m = etree.SubElement(xml, '{%s}Message' % ns_map['taxii'])
        m.text = self.message
    if self.subscription_information:
        xml.append(self.subscription_information.to_etree())
    for block in self.content_blocks:
        xml.append(block.to_etree())
    return xml

def to_dict(self):
    d = super(InboxMessage, self).to_dict()
    if self.message is not None:
        d['message'] = self.message
    if self.subscription_information:
        d['subscription_information'] = self.subscription_information.to_dict()
    d['content_blocks'] = []
    for block in self.content_blocks:
        d['content_blocks'].append(block.to_dict())
    return d

def to_text(self, line_prepend=''):  
    s = super(InboxMessage, self).to_text(line_prepend)
    s += line_prepend + '  Message: %s
' % self.message
    if self.subscription_information:
        s += self.subscription_information.to_text(line_prepend + STD_INDENT)
    s += line_prepend + '  Message has %s Content Blocks
' % len(self.content_blocks)
    for cb in self.content_blocks:
        s += cb.to_text(line_prepend + STD_INDENT)
    return s

@classmethod
def from_etree(cls, etree_xml):
    msg = super(InboxMessage, cls).from_etree(etree_xml)
    msg.message = get_optional_text(etree_xml, './taxii:Message', ns_map)
    subs_info = get_optional(etree_xml, './taxii:Source_Subscription', ns_map)

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

if subs_info is not None:
    msg.subscription_information = SubscriptionInformation.from_etree(subs_info)

content_blocks = etree_xml.xpath('./taxii:Content_Block', namespaces=ns_map)
msg.content_blocks = []
for block in content_blocks:
    msg.content_blocks.append(ContentBlock.from_etree(block))

return msg

@classmethod
def from_dict(cls, d):
    msg = super(InboxMessage, cls).from_dict(d)
    msg.message = d.get('message')
    msg.subscription_information = None
    if 'subscription_information' in d:
        msg.subscription_information = SubscriptionInformation.from_dict(d['subscription_information'])
    msg.content_blocks = []
    for block in d['content_blocks']:
        msg.content_blocks.append(ContentBlock.from_dict(block))

return msg

class SubscriptionInformation(TAXIIBase10):
    '''
    The Subscription Information component of a TAXII Inbox message.
    Arguments:
    feed_name (str): the name of the TAXII Data Feed from
    which this content is being provided. **Required**
    subscription_id (str): the Subscription ID for which this
    content is being provided. **Required**
    inclusive_begin_timestamp_label (datetime): a Timestamp Label
    indicating the beginning of the time range this Inbox Message
    covers. **Optional**
    inclusive_end_timestamp_label (datetime): a Timestamp Label
    indicating the end of the time range this Inbox Message
    covers. **Optional**
    '''

    def __init__(self, feed_name, subscription_id, 
                 inclusive_begin_timestamp_label, 
                 inclusive_end_timestamp_label):
        self._feed_name = feed_name
        self._subscription_id = subscription_id
        self._inclusive_begin_timestamp_label = inclusive_begin_timestamp_label
        self._inclusive_end_timestamp_label = inclusive_end_timestamp_label

    @property
def feed_name(self):
        return self._feed_name

    @feed_name.setter
def feed_name(self, value):
        do_check(value, 'feed_name', regex_tuple=uri_regex)
        self._feed_name = value

    @property
def subscription_id(self):
        return self._subscription_id

    @subscription_id.setter
def subscription_id(self, value):
        do_check(value, 'subscription_id', regex_tuple=uri_regex)

Jason Mack, jasonmack@gmail.com
self._subscription_id = value

@property
def inclusive_begin_timestamp_label(self):
    return self._inclusive_begin_timestamp_label

@inclusive_begin_timestamp_label.setter
def inclusive_begin_timestamp_label(self, value):
    value = check_timestamp_label(value, 'inclusive_begin_timestamp_label')
    self._inclusive_begin_timestamp_label = value

@property
def inclusive_end_timestamp_label(self):
    return self._inclusive_end_timestamp_label

@inclusive_end_timestamp_label.setter
def inclusive_end_timestamp_label(self, value):
    value = check_timestamp_label(value, 'inclusive_end_timestamp_label')
    self._inclusive_end_timestamp_label = value

def to_etree(self):
    xml = etree.Element('{%s}Source_Subscription' % ns_map['taxii'])
    xml.attrib['feed_name'] = self.feed_name
    xml.attrib['subscription_id'] = self.subscription_id
    ibtl = etree.SubElement(xml, '{%s}Inclusive_Begin_Timestamp' % ns_map['taxii'])
    ibtl.text = self.inclusive_begin_timestamp_label.isoformat()
    ietl = etree.SubElement(xml, '{%s}Inclusive_End_Timestamp' % ns_map['taxii'])
    ietl.text = self.inclusive_end_timestamp_label.isoformat()
    return xml

def to_dict(self):
    d = {}
    d['feed_name'] = self.feed_name
    d['subscription_id'] = self.subscription_id
    d['inclusive_begin_timestamp_label'] = self.inclusive_begin_timestamp_label.isoformat()
    d['inclusive_end_timestamp_label'] = self.inclusive_end_timestamp_label.isoformat()
    return d

def to_text(self, line_prepend=' '):
    s = line_prepend + '=== Subscription Information ===' + '
    s += line_prepend + '  Feed Name: %s
    s += line_prepend + '  Subscription ID: %s
    s += line_prepend + '  Incl. Begin TS Label: %s
    s += line_prepend + '  Incl. End TS Label: %s
    return s

@staticmethod
def from_etree(etree_xml):
    feed_name = etree_xml.attrib['feed_name']
    subscription_id = etree_xml.attrib['subscription_id']
    ibtl = parse_datetime_string(get_required(etree_xml, './taxii:Inclusive_Begin_Timestamp', ns_map).text)
    ietl = parse_datetime_string(get_required(etree_xml, './taxii:Inclusive_End_Timestamp', ns_map).text)
    return SubscriptionInformation(feed_name, subscription_id, ibtl, ietl)

@staticmethod
def from_dict(d):
    feed_name = d['feed_name']
    subscription_id = d['subscription_id']
    ibtl = parse_datetime_string(d['inclusive_begin_timestamp_label'])
    ietl = parse_datetime_string(d['inclusive_end_timestamp_label'])
    return SubscriptionInformation(feed_name, subscription_id, ibtl, ietl)

Jason Mack, jasonmack@gmail.com
class ManageFeedSubscriptionRequest(TAXIIIMessage):
    
    A TAXII Manage Feed Subscription Request message.

    Args:
    message_id (str): A value identifying this message.  **Required**
    extended_headers (dict): A dictionary of name/value pairs for
        use as Extended Headers.  **Optional**
    feed_name (str): the name of the TAXII Data Feed to which the
        action applies.  **Required**
    action (str): the requested action to take.  **Required**
    subscription_id (str): the ID of a previously created subscription.
        **Required** if `action==``:py:data:`ACT_UNSUBSCRIBE`, else
        **Prohibited**.
    delivery_parameters (list of DeliveryParameters): the delivery parameters
        for this request.  **Optional** Absence means delivery is not requested.

    message_type = MSG_MANAGE_FEED_SUBSCRIPTION_REQUEST

    def __init__(self, message_id, extended_headers=None,
                 feed_name=None, action=None, subscription_id=None,
                 delivery_parameters=None):
        super(ManageFeedSubscriptionRequest, self).__init__(message_id, extended_headers=extended_headers)
        self.feed_name = feed_name
        self.action = action
        self.subscription_id = subscription_id
        self.delivery_parameters = delivery_parameters

    @TAXIIIMessage.in_response_to.setter
    def in_response_to(self, value):
        if value:
            raise ValueError('in_response_to must be None')
        self._in_response_to = value

    @property
    def feed_name(self):
        return self._feed_name

    @feed_name.setter
    def feed_name(self, value):
        do_check(value, 'feed_name', regex_tuple=uri_regex)
        self._feed_name = value

    @property
    def action(self):
        return self._action

    @action.setter
    def action(self, value):
        do_check(value, 'action', value_tuple=ACT_TYPES)
        self._action = value

    @property
    def subscription_id(self):
        return self._subscription_id

    @subscription_id.setter
    def subscription_id(self, value):
        do_check(value, 'subscription_id', regex_tuple=uri_regex, can_be_none=True)
        self._subscription_id = value

    @property
    def delivery_parameters(self):
        return self._delivery_parameters

    @delivery_parameters.setter
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

def delivery_parameters(self, value):
    do_check(value, 'delivery_parameters', type=DeliveryParameters, can_be_none=True)
    self._delivery_parameters = value

def to_etree(self):
    xml = super(ManageFeedSubscriptionRequest, self).to_etree()
    xml.attrib['feed_name'] = self.feed_name
    xml.attrib['action'] = self.action
    if self.subscription_id is not None:
        xml.attrib['subscription_id'] = self.subscription_id
    if self.delivery_parameters:
        xml.append(self.delivery_parameters.to_etree())
    return xml

def to_dict(self):
    d = super(ManageFeedSubscriptionRequest, self).to_dict()
    d['feed_name'] = self.feed_name
    d['action'] = self.action
    d['subscription_id'] = self.subscription_id
    d['delivery_parameters'] = None
    if self.delivery_parameters:
        d['delivery_parameters'] = self.delivery_parameters.to_dict()
    return d

def to_text(self, line_prepend=''):  
    s = super(ManageFeedSubscriptionRequest, self).to_text(line_prepend)
    s += line_prepend + '  Feed Name: %s
' % self.feed_name
    s += line_prepend + '  Action: %s
' % self.action
    s += line_prepend + '  Subscription ID: %s
' % self.subscription_id
    if self.delivery_parameters:
        s += self.delivery_parameters.to_text(line_prepend + STD_INDENT)
    return s

@classmethod
def from_etree(cls, etree_xml):
    kwargs = dict(
        feed_name = get_required(etree_xml, './@feed_name', ns_map),
        action = get_required(etree_xml, './@action', ns_map),
        # subscription_id is not required for action 'SUBSCRIBE'
        subscription_id = get_optional(etree_xml, './@subscription_id', ns_map),
    )
    # marked as required in spec but as optional is XSD
    delivery = get_optional(etree_xml, './taxii:Push_Parameters', ns_map)
    if delivery is not None:
        kwargs['delivery_parameters'] = DeliveryParameters.from_etree(delivery)
    msg = super(ManageFeedSubscriptionRequest, cls).from_etree(etree_xml, **kwargs)
    return msg

@classmethod
def from_dict(cls, d):
    kwargs = dict(
        feed_name = d['feed_name'],
        action = d['action'],
        subscription_id = d['subscription_id'],
        delivery_parameters = DeliveryParameters.from_dict(d['delivery_parameters'])
    )
    msg = super(ManageFeedSubscriptionRequest, cls).from_dict(d, **kwargs)
    return msg

class ManageFeedSubscriptionResponse(TAXIIMessage):

    def __init__(self, *args, **kwargs):
        self.data = kwargs.pop('data', None)
        super().__init__(*args, **kwargs)

    def __dict__(self):
        return {k: getattr(self, k) for k in self.__slots__ if not k.startswith('_')}

    def _to_etree(self):
        xml = ET.Element('manage_feed_subscription_response')
        xml.attrib['data'] = self.data
        for slot in self.__slots__:
            if not slot.startswith('_') and hasattr(self, slot):
                xml.append(getattr(self, slot).to_etree())
        return xml

    def to_etree(self):
        return self._to_etree()

    def _to_dict(self):
        return {k: getattr(self, k) for k in self.__slots__ if not k.startswith('_')}

    def to_dict(self):
        return self._to_dict()
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Args:
message_id (str): A value identifying this message. **Required**
in_response_to (str): Contains the Message ID of the message to
which this is a response. **Required**
extended_headers (dict): A dictionary of name/value pairs for
use as Extended Headers. **Optional**
feed_name (str): the name of the TAXII Data Feed to which
the action applies. **Required**
message (str): additional information for the message recipient.
**Optional**
subscription_instances (list of SubscriptionInstance): **Optional**

message_type = MSG_MANAGE_FEED_SUBSCRIPTION_RESPONSE

def __init__(self, message_id, in_response_to, extended_headers=None,
            feed_name=None, message=None, subscription_instances=None):
    super(ManageFeedSubscriptionResponse, self).__init__(message_id, in_response_to, extended_headers=extended_headers)
    self.feed_name = feed_name
    self.message = message
    self.subscription_instances = subscription_instances or []

@TAXIIMessage.in_response_to.setter
def in_response_to(self, value):
    do_check(value, 'in_response_to', regex_tuple=uri_regex)
    self._in_response_to = value

@property
def feed_name(self):
    return self._feed_name

@feed_name.setter
def feed_name(self, value):
    do_check(value, 'feed_name', regex_tuple=uri_regex)
    self._feed_name = value

@property
def subscription_instances(self):
    return self._subscription_instances

@subscription_instances.setter
def subscription_instances(self, value):
    do_check(value, 'subscription_instances', type=SubscriptionInstance)
    self._subscription_instances = value

def to_etree(self):
    xml = super(ManageFeedSubscriptionResponse, self).to_etree()
    xml.attrib['feed_name'] = self.feed_name
    if self.message is not None:
        m = etree.SubElement(xml, '{%s}Message' % ns_map['taxii'])
        m.text = self.message
        for subscription_instance in self.subscription_instances:
            xml.append(subscription_instance.to_etree())
    return xml

def to_dict(self):
    d = super(ManageFeedSubscriptionResponse, self).to_dict()
    d['feed_name'] = self.feed_name
    if self.message is not None:
        d['message'] = self.message
    d['subscription_instances'] = []
    for subscription_instance in self.subscription_instances:
        d['subscription_instances'].append(subscription_instance.to_dict())
    return d

Jason Mack, jasonmack@gmail.com
def to_text(self, line_prepend=''):  
s = super(ManageFeedSubscriptionResponse, self).to_text(line_prepend)  
s += line_prepend + '  Feed Name: %s
' % self.feed_name  
s += line_prepend + '  Message: %s
' % self.message  
for si in self.subscription_instances:  
    s += si.to_text(line_prepend + STD_INDENT)  
return s

@classmethod
def from_etree(cls, etree_xml):
    kwargs = {}  
kwargs['feed_name'] = etree_xml.attrib['feed_name']  
kwargs['message'] = get_optional_text(etree_xml, '/taxii:Message', ns_map)  
kwargs['subscription_instances'] = []  
for si in etree_xml.xpath('/taxii:Subscription', namespaces=ns_map):  
    kwargs['subscription_instances'].append(SubscriptionInstance.from_etree(si))  

    msg = super(ManageFeedSubscriptionResponse, cls).from_etree(etree_xml, **kwargs)  
    return msg

@classmethod
def from_dict(cls, d):
    kwargs = {}  
kwargs['feed_name'] = d['feed_name']  
kwargs['message'] = d.get('message')  
kwargs['subscription_instances'] = []  
for instance in d['subscription_instances']:
    kwargs['subscription_instances'].append(SubscriptionInstance.from_dict(instance))  
    msg = super(ManageFeedSubscriptionResponse, cls).from_dict(d, **kwargs)  
    return msg

class SubscriptionInstance(TAXIIBase10):

    The Subscription Instance component of the Manage Feed Subscription Response message.

    Args:
        subscription_id (str): the id of the subscription. **Required**
        delivery_parameters (DeliveryParameters): the parameters
            for this subscription. **Required** if responding to message
            with 'action'=='ACT_STATUS', otherwise **Prohibited**
        poll_instances (list of PollInstance): Each Poll
            Instance represents an instance of a Poll Service that can be
            contacted to retrieve content associated with the new
            Subscription. **Optional**

    def __init__(self, subscription_id, delivery_parameters=None, poll_instances=None):
        self.subscription_id = subscription_id  
        self.delivery_parameters = delivery_parameters  
        self.poll_instances = poll_instances or []

    @property
    def sort_key(self):
        return self.subscription_id

    @property
    def subscription_id(self):
        return self._subscription_id

    @subscription_id.setter
    def subscription_id(self, value):
        self._subscription_id = value

Jason Mack, jasonmack@gmail.com
do_check(value, 'subscription_id', regex_tuple=uri_regex)
    self._subscription_id = value

@property
def delivery_parameters(self):
    return self._delivery_parameters

@delivery_parameters.setter
def delivery_parameters(self, value):
    do_check(value, 'delivery_parameters', type=DeliveryParameters, can_be_none=True)
    self._delivery_parameters = value

@property
def poll_instances(self):
    return self._poll_instances

@poll_instances.setter
def poll_instances(self, value):
    do_check(value, 'poll_instances', type=PollInstance, can_be_none=False)
    self._poll_instances = value

def to_etree(self):
    xml = etree.Element('{%s}Subscription' % ns_map['taxii'])
    xml.attrib['subscription_id'] = self.subscription_id
    if self.delivery_parameters:
        xml.append(self.delivery_parameters.to_etree())
    for poll_instance in self.poll_instances:
        xml.append(poll_instance.to_etree())
    return xml

def to_dict(self):
    d = {
        'subscription_id': self.subscription_id
    }
    if self.delivery_parameters:
        d['delivery_parameters'] = self.delivery_parameters.to_dict()
    else:
        d['delivery_parameters'] = None
    d['poll_instances'] = []
    for poll_instance in self.poll_instances:
        d['poll_instances'].append(poll_instance.to_dict())
    return d

def to_text(self, line_indent=''):  
    s = line_indent + '=== Subscription Instance ===
    s += line_indent + ' Subscription ID: %s' % self.subscription_id
    if self.delivery_parameters:
        s += self.delivery_parameters.to_text(line_indent + STD_INDENT)
    for pi in self.poll_instances:
        s += pi.to_text(line_indent + STD_INDENT)
    return s

@staticmethod
def from_etree(etree_xml):
    subscription_id = etree_xml.attrib['subscription_id']
    _delivery_parameters = get_optional(etree_xml, './taxii:Push_Parameters', ns_map)
    if _delivery_parameters:
        delivery_parameters = DeliveryParameters.from_etree(_delivery_parameters)
    else:
        delivery_parameters = None
    poll_instances = []
    for poll_instance in etree_xml.xpath('./taxii:Poll_Instance', namespaces=ns_map):
        poll_instances.append(PollInstance.from_etree(poll_instance))
    return Subscription(subscription_id, delivery_parameters, poll_instances)
poll_instances.append(PollInstance.from_etree(poll_instance))

return SubscriptionInstance(subscription_id, delivery_parameters, poll_instances)

@staticmethod
def from_dict(d):
    subscription_id = d['subscription_id']

    if d.get('delivery_parameters'):
        delivery_parameters = DeliveryParameters.from_dict(d['delivery_parameters'])
    else:
        delivery_parameters = None

    poll_instances = []
    for poll_instance in d['poll_instances']:
        poll_instances.append(PollInstance.from_dict(poll_instance))

    return SubscriptionInstance(subscription_id, delivery_parameters, poll_instances)

class PollInstance(TAXIIBase):
    
    The Poll Instance component of the Manage Feed Subscription Response message.

    Args:
        poll_protocol (str): The protocol binding supported by this instance of a Polling Service. **Required**
        poll_address (str): the address of the TAXII Daemon hosting this Poll Service. **Required**
        poll_message_bindings (list of str): one or more message bindings that can be used when interacting with this Poll Service instance. **Required**

    def __init__(self, poll_protocol, poll_address, poll_message_bindings=None):
        self.poll_protocol = poll_protocol
        self.poll_address = poll_address
        self.poll_message_bindings = poll_message_bindings or []

    @property
def sort_key(self):
        return self.poll_address

    @property
def poll_protocol(self):
        return self.poll_protocol

    @poll_protocol.setter
def poll_protocol(self, value):
        do_check(value, 'poll_protocol', regex_tuple=uri_regex)
        self.poll_protocol = value

    @property
def poll_message_bindings(self):
        return self.poll_message_bindings

    @poll_message_bindings.setter
def poll_message_bindings(self, value):
        do_check(value, 'poll_message_bindings', regex_tuple=uri_regex)
        self.poll_message_bindings = value

    def to_etree(self):
        xml = etree.Element('{%s}Poll_Instance' % ns_map['taxii'])
        pb = etree.SubElement(xml, '{%s}Protocol_Binding' % ns_map['taxii'])
        pb.text = self.poll_protocol

Jason Mack, jasonmack@gmail.com
Using Network Based Security Systems to Search for STIX and TAXII Based Indicators of Compromise

Jason Mack, jasonmack@gmail.com

```python
a = etree.SubElement(xml, '{%s}Address' % ns_map['taxii'])
a.text = self.poll_address

for binding in self.poll_message_bindings:
    b = etree.SubElement(xml, '{%s}Message_Binding' % ns_map['taxii'])
    b.text = binding

return xml

def to_dict(self):
    d = {}
    d['poll_protocol'] = self.poll_protocol
    d['poll_address'] = self.poll_address
    d['poll_message_bindings'] = []
    for binding in self.poll_message_bindings:
        d['poll_message_bindings'].append(binding)

    return d

def to_text(self, line_prepend=''):  # Add line_prepend=""
    s = line_prepend + '=== Poll Instance ===
    s += line_prepend + '  Protocol Binding: %s
    s += line_prepend + '  Address: %s
    for mb in self.poll_message_bindings:
        s += line_prepend + '  Message Binding: %s
    return s

@staticmethod
def from_etree(etree_xml):
    poll_protocol = get_required(etree_xml, './taxii:Protocol_Binding', ns_map).text
    address = get_required(etree_xml, './taxii:Address', ns_map).text
    poll_message_bindings = []
    for b in etree_xml.xpath('./taxii:Message_Binding', namespaces=ns_map):
        poll_message_bindings.append(b.text)

    return PollInstance(poll_protocol, address, poll_message_bindings)

@staticmethod
def from_dict(d):
    return PollInstance(**d)
```

# EVERYTHING BELOW HERE IS FOR BACKWARDS COMPATIBILITY#

# Add top-level classes as nested classes for backwards compatibility
DiscoveryResponse.ServiceInstance = ServiceInstance
FeedInformationResponse.FeedInformation = FeedInformation
FeedInformation.PollingServiceInstance = PollingServiceInstance
FeedInformation.SubscriptionMethod = SubscriptionMethod
ManageFeedSubscriptionResponse.PollInstance = PollInstance
ManageFeedSubscriptionResponse.SubscriptionInstance = SubscriptionInstance
InboxMessage.SubscriptionInformation = SubscriptionInformation

# Constants not imported in `from constants import *`
MSG_TYPES = MSG_TYPES_10
ST_TYPES = ST_TYPES_10
ACT_TYPES = ACT_TYPES_10
SVC_TYPES = SVC_TYPES_10

from common import (generate_message_id)

Jason Mack, jasonmack@gmail.com

© 2015 The SANS Institute

Author retains full rights.
<table>
<thead>
<tr>
<th>SANS OnDemand</th>
<th>OnlineUS</th>
<th>Anytime</th>
<th>Self Paced</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS SelfStudy</td>
<td>Books &amp; MP3s OnlyUS</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>