Content Security Framework

Sreenu Pillutla, Director of Engg.
McAfee, Inc.
Sreenu_Pillutla@mcafee.com

May 25, 2013

Agenda

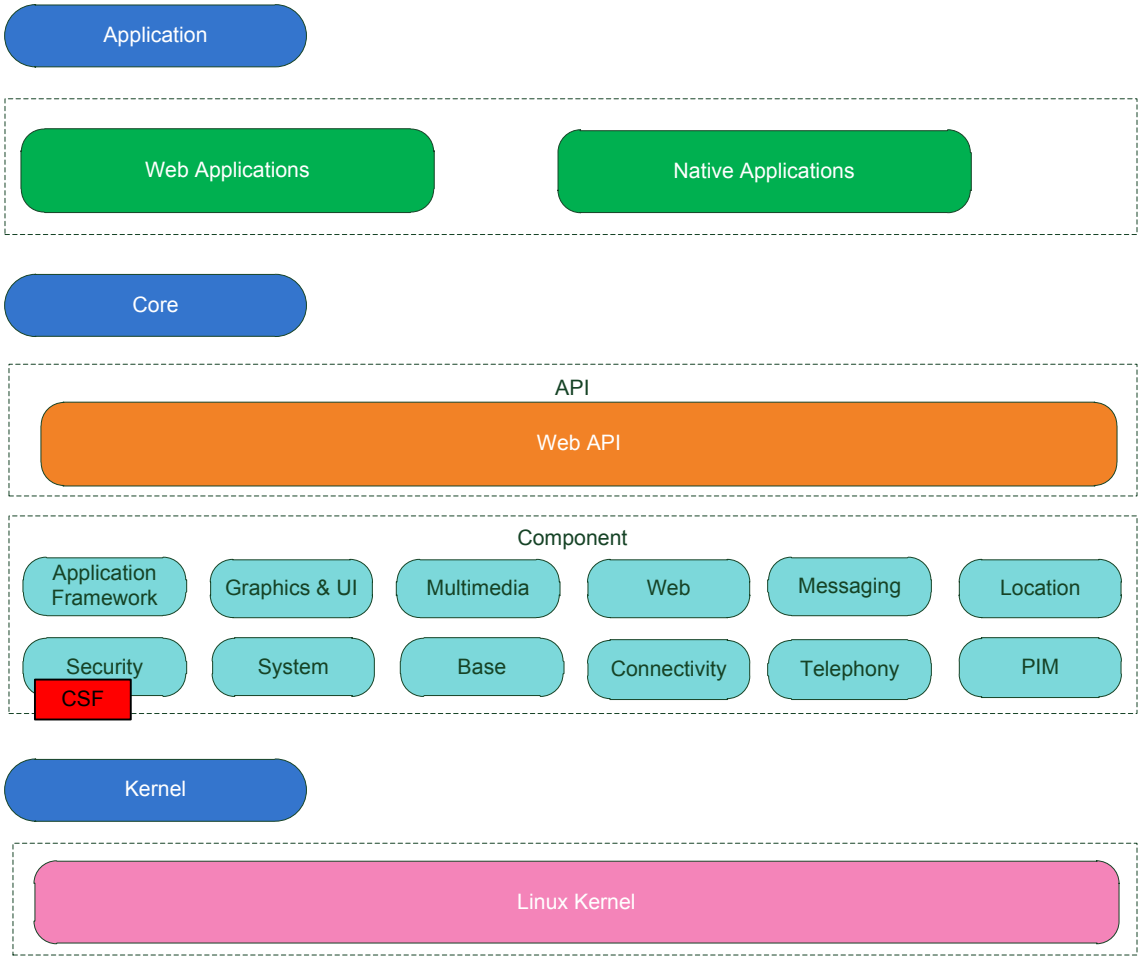
- Why Content Security Framework (CSF)
- What is CSF?
  - Details, Design Concepts
- How to implement CSF
  - Security Engines
    - Scan Engine
      - API, Usage
    - Site Engine
      - API, Usage
- Recap

Why CSF?

- Malware threats are increasing in numbers and complexity, nothing exists as a basis of the platform –> CSF allows platform **hardening**
  - Helps combat malware, protects web browsing
- Traditional methods do everything in **app** layer. Depend on hooks, events which are not **reliable**, cause **performance** fears, closed OS does not allow
- Low level hooks and heavy weight solutions -> **Efficient** solutions for mobile
- Security for modern operating systems is around access control and vulnerabilities. Solutions galore, does not address human errors. E.g. Clicking compelling URLs, leading to phishing websites -> Provide more **secure** OS

Platform hardening, secure, efficient, reliable

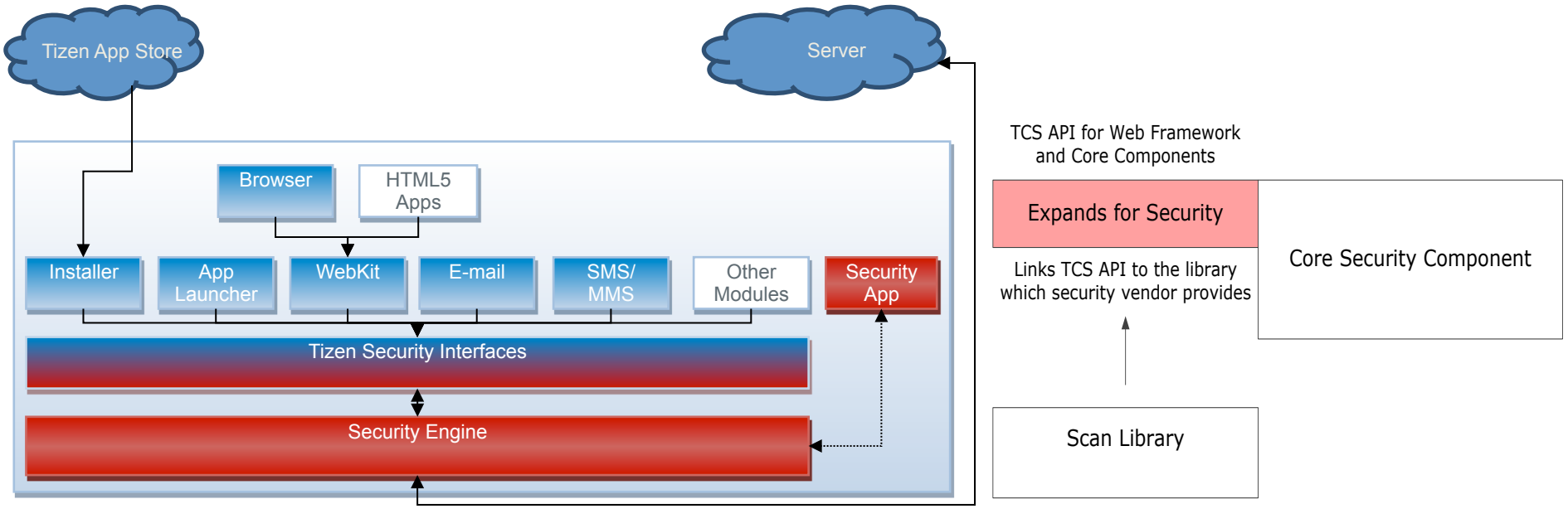
What is CSF?



- CSF is a part of security component of Tizen stack. Expands security component, framework to enhance OS Content security, not AM app
- Set of APIs that can be used by other components to **scan content** and **URLs**.
- CSF is responsible for plug-in management. Error Handling when there is no plugin
- The concrete implementation of scanning is provided by 'Content Security Plug-in's (**engines**).

CSF -> APIs, Test Kits, Stub Engines, Documentation, Plugin management

What is CSF?



- Open interfaces accessed by OS and by applications
- System applications (browser, installer, mailer, messenger) call framework API to scan the data and URLs. Framework will pass calls to plugin (if exists) to scan data, URLs.
- System installer will create symbol link to the security plugins so that framework can load plugin to scan data passed down by system applications.
- Vendors
  - security **engines** that implement the interfaces
  - mobile **research** on Tizen provides malware and reputation services

Open Interfaces for OS/apps, Interfaces to Vendor apps

CSF Design Concepts

- Light weight security implementation -> **OS** friendly
  - **Enables** system applications to initiate scan, no overhead of hooking into file system or low level protocol stack
  - System applications can scan **when they want**, they are aware of content, data type, and timing (Scan when needed)

- Action driven -> **User** friendly
  - Based on **user action**, system applications know **when to scan**, can balance security and performance

- Standard Interfaces -> **Vendor** friendly
  - **Standardized** interface to system applications and security vendor plugins
  - Vendors **provide** security solutions conforming to CSF

Light Weight, Action Driven, Standard Interfaces

How to implement CSF?

- Security API to upper layer
  - System apps call security API to scan the content that is passed from the web applications or core components
  - uses stub library if security plugin is not available
- Provides interface for vendors
  - Security engines link to security framework library dynamically
- Security Engines include Scan Engine and Site Engine

Interfaces to Vendors, Security API to upper layer Apps

CSF - Security Engines

Scanning Engine

- Malware Scanning, Content Inspection
- Scans Apps, Files, Content

Site Engine

- Scans URLs
- Reputation of one or more URL
- Category for one or more URL
- Policies can be defined
- URLs can be blocked, redirected to a block page

Engines – Scan, Site

Security Engines

- Security Engine has two parts:
  - Scan Engine: content security agent
    - Provides ability of scan the **content** (e.g., file or data on the memory)
    - **Detects** malware, virus, trojans, spam, etc.
    - Connects to the vendor cloud for detection data (vendor specific)
  - Site Engine: URL security agent
    - Provides individual URL's security information
      - **Reputation**, content **categories**
    - Provides ability to create own access **policy**
    - Allow to **block** URL access and redirect to blocking notification page

Engines – Scan, Site

Scan Engine

- Scan Engine implementation – Specific to Vendor
- Content/context sensitive analysis and inspection
  - applications control **when** the scanner is called (e.g. before rendering HTML page, before executing JavaScript)
  - applications provide the scan engine with **preprocessed** data with type/ characteristics to the scanner
  - **Advantages** over traditional security scanning
    - No low level file hooks are needed
  - Content **data type** examples:
    - HTML
    - URL
    - Text
    - JavaScript

Specific to Vendor, Application has control

Scan Engine - Cooperative Scanning

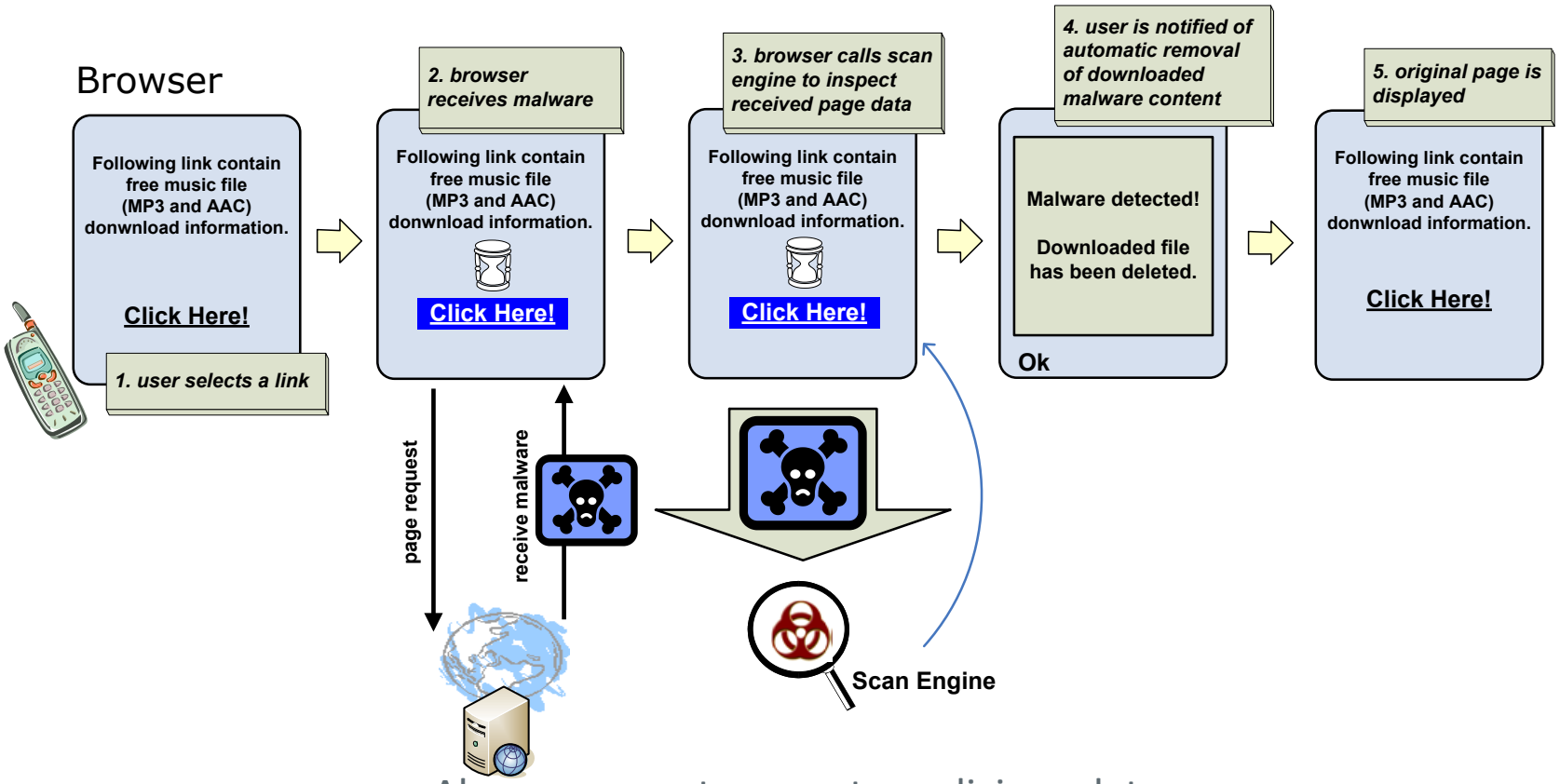
- **Integrated** in caller application/module
  - Caller application implements scan API
  - Caller handles the case of infected content
    - Shows 'blocked infected content' message, etc.

- Possible to specify content **data type** for scanning
  - Optimizes proper scan timing during content handling

- Better **user experience** for notifying detection
  - Application/content specific notify method, message, timing, handling

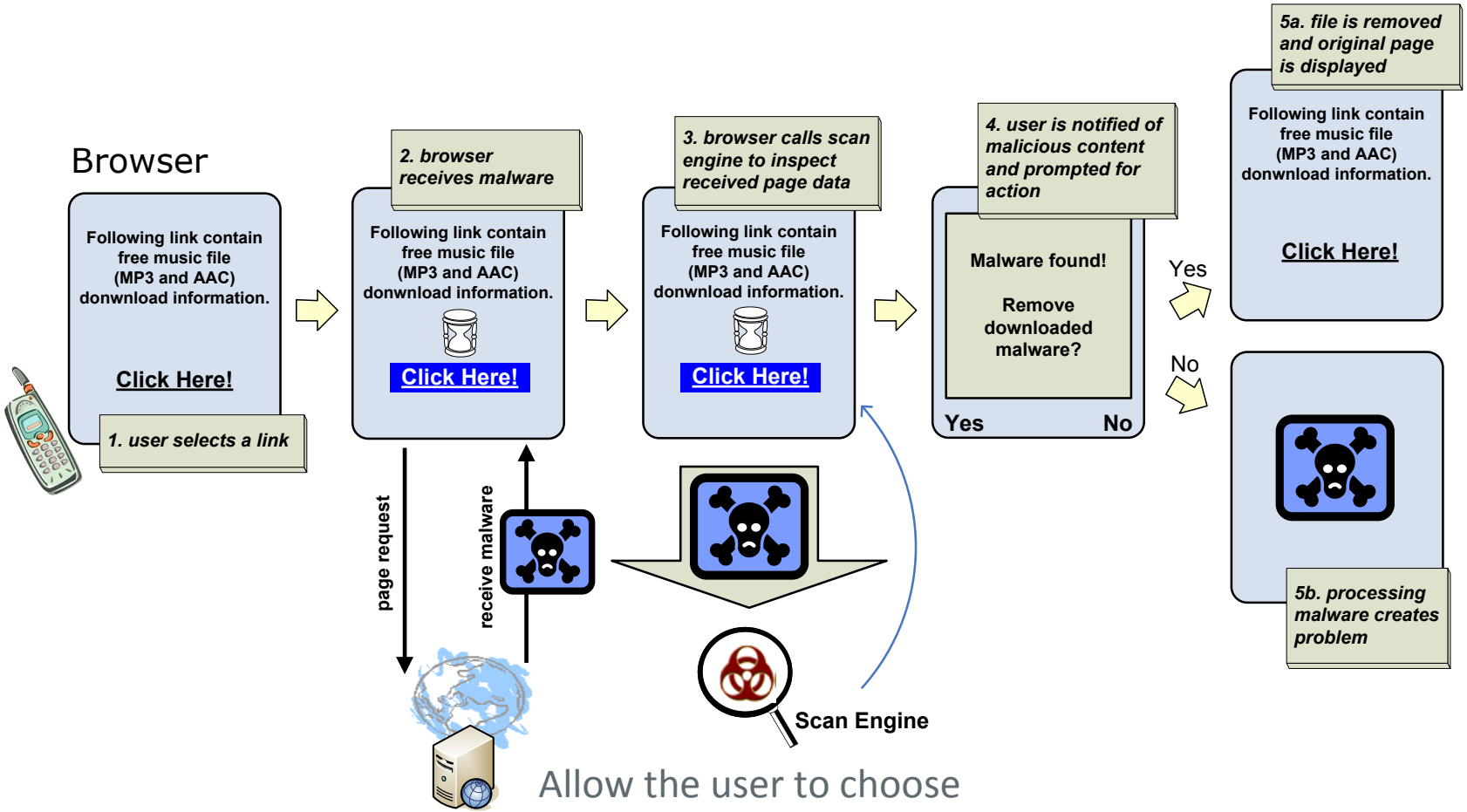
Integrated in app, specify data type, better UX

Cooperative Scanning Example



Always prevent access to malicious data

Cooperative Scanning Example (cont'd)



Cooperative Scanning Use Cases

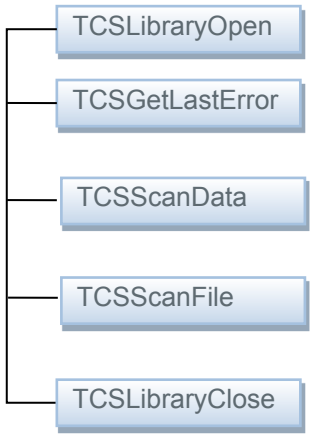
Caller	Data to be scanned	Data type	After receiving	Before receiving	Before rendering	Before invoking	Before connect
Email client	URL, HTML, Email body	Text, HTML, Email	O	O	R		
Browser	HTML, JavaScript, embedded text (USSD)	Text, HTML, JavaScript	O	O	R		O
Installer	HTML5	HTML	O	O		R	
Application Launcher	HTML5	HTML				R	
Messenger	SMS/MMS	Text, HTML	O	O		R	

R: recommended

O: optional

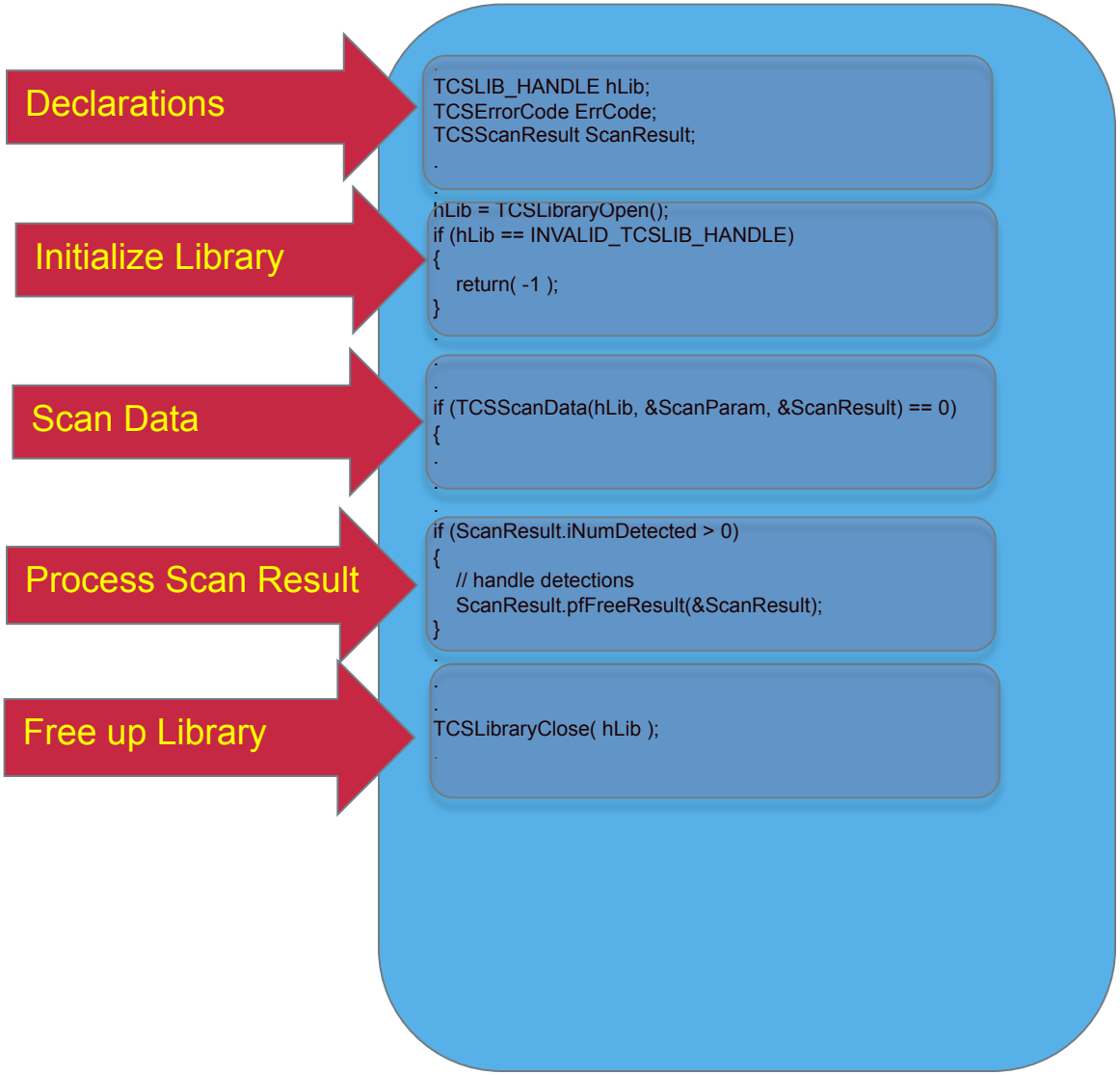
Scan Engine API

- Initialization
  - **`TCSLibraryOpen/TCSLibraryClose`**
    - Initializes/closes a library handle
- Error retrieval
  - **`TCSGetLastError`**
    - Returns the library instance's last-error code
- Content scanning
  - **`TCSScanData`**
    - Scans application data for malware [Content in memory]
  - **`TCSScanFile`**
    - Scans file for malware [Content on permanent storage]



Scan Engine API - Usage

- Declare
- Initialize Library
- Scan Data
- Process Result
- Close Library



Site Engine API - Usage

- Distributed as library file and a header file
- Include the header file in source file and link with the library
- Initialize web protection lib
- Create web protection config
- Create web protection HTTP request call back functions
- Create web protection policy
- Check URL against the cloud
- Retrieve rating of the URL
- Check any violations against the local policy
- Un-initialize

Initiate Library

```
//Init Lib
TWPAPIInit tInit;
tInit.api_version = TWP_API_VERSION;
tInit.memallocfunc = malloc;
tInit.memfreefunc = free;
TWPLIB_HANDLE hLib = TWPInitLibrary( &tInit );
```

Create Configuration

```
//Create config
TWPConfigurationHandle hCfg;
TWPConfiguration tCfg;
tCfg.config_version = TWP_CONFIG_VERSION;
tCfg.client_id = "<your ID provided by security vendor>";
tCfg.client_key = "< Key provided by security vendor>";
tCfg.obfuscate_request = 1;
tCfg.randomfunc = random;
...
TWPConfigurationCreate( hLib, &tCfg, &hCfg );
```

Create Policy

```
//Create Policy
TWPPolicyCreate( hLib, MyCategories, NumOfCategories,
&hPolicy);
```

Lookup URL

```
//Lookup URLs
TWPLookupUrls(hLib, hCfg, &MyRequest, 1, Urls, NumOfUrls,
                    &hResponse);
```

Check violation against the policy

```
//Get URL Rating, the first one
TWPResponseGetUrlRatingByIndex(hLib, hResponse, 0, &hRating);

//Validate Policy
TWPPolicyValidate(hLib, hPolicy, hRating, &iViolated);

if (iViolated) { // violated
// block access
} else { // not violated
}
```

Site Engine API - Usage

```
void webkit_web_view_load_html_string(WebKitWebView *web_view, const gchar *content,
                                      const gchar *base_uri)
{
    const char *Urls[1];
    TWPResponseHandle hResponse;
    TWPRating hRating;
    int violated = 0;

    Urls[0] = asprintf(&Urls[0], "%s/%s", base_url, content);
    TWPLookupUrls(hLib, mTwpConfig, mRequest, 0, Urls, 1, &hResponse);
    TWPResponseGetUrlRatingByIndex(hLib, hResponse, 0, &hRating);
    TWPPolicyValidate(hLib, mPolicy, hRating, &violated);

    if (violated) {
        show_block_page();
    } else {
        Original_webkit_web_view_load_html_string(web_view, content, base_uri);
    }
}
```

Filtering Bad URL

Access Good URL

Site Engine - How to create policy?

```
TWPCategories FilteredCategories[] = {
                Alcohol,
                Gambling,
                Maliciousssites,
                Pornography,
                Spywareadwarekeyloggers,
                ... ...
};
```

Add the pre-defined categories you want to block

```
TWPPolicyCreate(hLib, FilteredCategories,
                FilteredCategories / sizeof(TWPCategories),
                &hPolicy);
```

Create your policy with sites categories

- HTTP request **callback** functions are used by web protection library.
  - HTTP stack **independent**.
  - Web protection library allows caller to **customize** their own HTTP call rather than forcing standard HTTP calls on different platforms.
    - For example, some callers might want to add proxy to their HTTP call.

Site Engine - Browser Redirection

- Vendors can show value to users by intercepting navigation to dangerous sites
- Complete control over how user experience is enhanced

CSF - Recap

- Why CSF
  - **Hardens** Platform, **Secure**, **Efficient**, **Reliable**

- What is CSF
  - **Extends** Security Components
  - **Contains** APIs, Test Kits, Stub Engines, Documentation
  - **Provides** Plugin management
  - Design **Concepts** – Light Weight, Action Driven, Standard Interfaces

- How to implement CSF?
  - System Apps call CSF for **scanning** URLs and Data which in turn load security engines
  - Engines **implement** interfaces, perform data, URL scans
  - Security Engines
    - Scan Engine - APIs, Usage
    - Site Engine – APIs, Usage

QUESTIONS