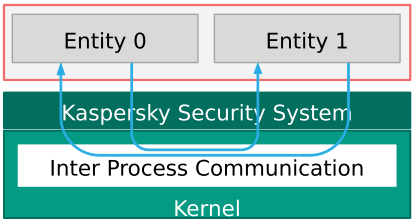
kaspersky BRING ON THE FUTURE

Kaspersky Security System Technical Data

Kaspersky Security System

KSS can be integrated into the existing environment to intercept sensitive operations, validate the correctness of message structures and implement an access control policy that can allow or block operations.



The main KSS objective is to provide a tool to support higher level security policies in a flexible way. In particular it means the following:

- Flexible and extendable access control mechanism.
- A set of security policy implementations.
- SDK to add new policy implementations.

Kaspersky Security System (KSS) is a security framework that enhances an existing operational environment with access control features.

KSS can be run on top of a system implementing separation kernel (SK) architecture. KasperskyOS is an example of such a system: it is a microkernel OS providing strict domain separation with IPC as the only mechanism available for domains to communicate.

While SK provides a good basis for a secure solution, it is not sufficient in many practical cases where complex components with different trust levels communicate and enforcement of diverse security properties is required.

Additional means need to be implemented above the separation kernel architecture to specify and enforce higher level security policies. Any OS provides its own means to express security properties; the problem is that they are typically fixed and therefore such an approach is limited.

The more complex the solution, the more demand there is to express and enforce diverse security properties. It is virtually impossible to achieve this within a fixed security model.

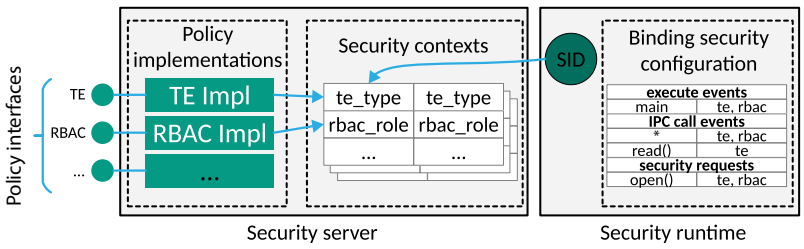
It becomes especially important when an untrusted third party is involved in solution development or open source components are used, and it can increase the possibility of vulnerabilities discovery. If one could express the security properties, it would be possible to constrain such components and make the solution more secure.

That is why there should be a flexible way to specify and enforce higher level security policy that is:

- **Reusable:** applicable to a wide range of applications.
- **Composable:** compose system-wide security configuration from smaller independent policies.
- **Higher level:** declarative, expressible in business-domain terms rather than OS-specific notions.
- **Extendable:** provide means to add new types of policies.

Architecture

KSS logically comprises two components: KSS Runtime (Security Runtime) and Security Server, where Security Server operates solely over abstract security domains, while KSS Runtime acts as a glue layer between business logic (applications) and security logic (security policies).



KSS Runtime:
- keeps a binding between system interactions and security rules;
- requests Security Server to compute those polices;
- combines the results of computations into access decisions.

Security Server:
- provides implementations of all policies;
- manages security contexts;
- serves requests from Security Runtime.

Security as a Separable Concern

KSS is designed around a concept of security as a separable concern. This concept assumes the security enforcement mechanism is separated from business logic. It is a crucial part of achieving flexibility.

With an approach that treats security as a separable concern, we get the following advantages:

For business applications:
- no need for applications to implement security policies;
- no need to change applications if a security policy changes;
- security policies are not limited to the tools supported by applications.

For security policies:
- policies are abstracted away from applications;
  - policies operate over abstract domains;
  - policies are not aware of differences between applications, resources, etc;
- policy can remain stable even if applications change significantly;
- system-wide security policy is a composition of smaller policies.

But in order to achieve flexibility more features are required. In particular, KSS relies on 'typed communications'.

Typed Communications

Every time KSS is given a message to control, it can use its type information to apply a policy. But prior to any policy invocation, KSS checks the message format and ensures that this message is well formed according to the declared interface. This kind of check ensures that applications exchange messages in the correct format. It illuminates the source of many dangerous attacks with intentionally invalid data requests.

Some security policies require knowledge of message structure. For such policies, just knowing an interaction occurs is insufficient.

For example, a service may provide an interface to set/get parameters for a technological process. And there is a safety policy that enforces a parameter (e.g. desired temperature) that cannot be set outside a predefined range and/or it may depend on previously observed values.

For a reference monitor a message is just a vector of bytes. It knows nothing about the meaning of those bytes per se.

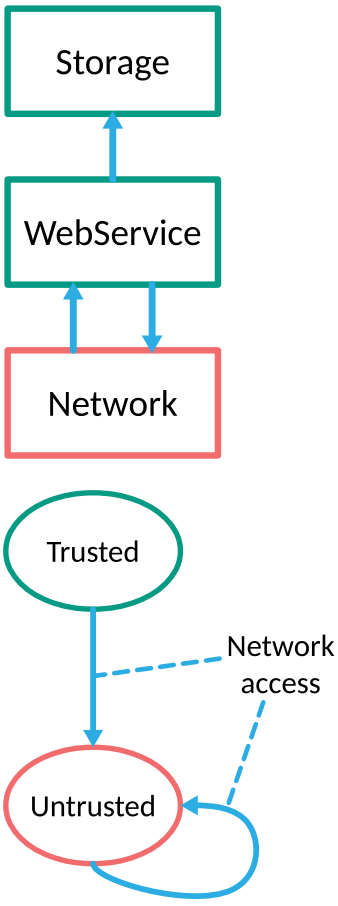
To provide knowledge about the message structure (and meaning) to KSS, the following approach is adopted: applications must declare their interfaces, and KSS SDK provides Interface Definition Language (IDL) to define interfaces and data structures. Every entity (business application, device driver, etc.) must statically declare all interfaces it provides as a service.

```
package filesys

typedef sequence<Char, 255> Path;
typedef sequence<UInt8, 4096> Buffer;
typedef SInt32 Result;

interface IFileSystem {
    open(in Path name, in UInt32 mode, out Handle fd);

    read(in Handle fd, in UInt32 pos, in UInt32 len,
        out Buffer buf, out Result retcode);

    write(in Handle fd, in UInt32 pos, out UInt32 len,
        in Buffer buf, out Result retcode);
}
```

A security context is associated with every subject (and object) and serves two purposes:

- It uniquely identifies the security domain.
- It keeps the state that is needed by stateful policies to calculate decisions.

When a policy is analyzing communication, it may use the security contexts of the communicating parties. A security context is identified by an integral descriptor called a security identifier, or SID.



Policy Specification Language

Interfaces are collections of methods that comprise in and out arguments. This definition implies that every method describes two separate messages: request (in-message) and response (out-message). So every synchronous (request/reply) interaction is turned into a method call.

KSS has the ability to bind a particular policy to every interaction in the system, using the knowledge about what method call this message represents. To specify such bindings, KSS provides Security Configuration Language (CFG). The key feature of this language is the ability to compose different security policies in a higher level way.

```
01 entity Application;
02
03 execute dst = Application {
04        rbac.inheritRole (src, dst);
05 }
06
07 request dst = Application, message = IStatus.getStats {
08        rbac.checkPermissions ["GetStats"] (src);
09 }
```

Line 01 declares that there is an entity named **Application** in the system. The configuration compiler uses this information to get information about all the interfaces **Application** implements. Note that there may be more than one instance of the same entity **Application**.

Lines 03-05 specify the first binding rule. It says: whenever an entity (with SID **src**) executes **Application** (SID **dst**), apply security policy **rbac.inheritRole (src, dst)**. This policy assigns src's role to a newly created instance of **Application** with the identifier **dst**. Note that binding rules have two predefined SIDs: initiator (or source) src, and recipient (or destination) **dst**.

Lines 07-09 specify another binding rule. In this case, it says: whenever an entity **(src)** requests (or calls) from **Application (dst)** method **getStatus** of interface **IStatus,** apply policy **rbac.checkPermissions.**

Line 08 contains a policy call with static configuration **["GetStats"]**. This is just a JSON array of permissions required to get access to the method. Line 08 also states that the policy argument is src. It means that **rbac.checkPermissions** must take the role associated with **src** and ensure that this role has permission **["GetStats"]**.

Example: WebService

A web service **(WebService)** application has access to a system configuration database **(Storage)** to get its initial configuration. It also has access to a network subsystem **(Network)** to serve requests from remote clients.

It is assumed that Storage may contain sensitive data. However, **Network** is considered an untrusted component.

- **Storage** – is a configuration database with sensitive data; it is considered a **trusted** component.
- **Network** – is a communication component to interact with the public network; it is considered an **untrusted** component.
- **WebService** – is an application that needs access to Storage to get startup configuration and access to **Network** to do its job.

WebService is considered a **trusted** component until it sends the first request to **Network** (thus it can have access to **Storage** at the beginning). As soon as WebService interacts with **Network** it is no longer **trusted**.

This simple security property can be formalized in the following way: the system can be in two different states: initial (or **Trusted**) and **Untrusted**. As soon as **WebService** gets data from **Network**, there can only be one transition – from a **Trusted** state to **Untrusted**.

At the beginning WebService has a trusted state.
As soon as it accesses Network for the first time it becomes untrusted.
There is no way to become trusted again.

```
01 use family example_model = flow {
02         states: [ trusted, untrusted ],
03         initial: trusted,
04         transitions: {
05           trusted: [ untrusted ],
06           untrusted: [ untrusted ]
07         }
08 };
09
10 execute dst=WebService { example_model.restart; }
11
12 request src=WebService, dst=Storage {
13                 example_model.allow [ "trusted" ];
14 }
15
16 request src=WebService,
17                   dst=Network {
18                 example_model.enter "untrusted";
19 }
```

KSS is capable of specifying extremely complex security properties to meet real-life requirements.

Lines 01-08 introduce a definition of the model as an unlabeled transition system using the **flow** policy family. The model is referenced below with name **example_model**. **Flow** is one of the policies provided with KSS SDK.

Line 02 specifies the set of states (two of them).

Line 03 specifies the initial state **(trusted)**.

Lines 04-07 specify permitted transitions within the system. It shows that there is no transition back to a **trusted** state.

Line 10 specifies a binding rule. As soon as **WebService** is executed, it sets the model to its initial state (because **WebService** is supposed to be trusted upon execution).

Lines 12-14 specify a **request** binding rule. Upon any request from **WebService** to **Storage**, it applies the policy **example_model.allow**, which checks that the system is in a **trusted** state; otherwise, it prohibits access.

Lines 16-19 specify another **request** binding rule. Upon any request from **WebService** to **Network**, it applies the policy **example_model.enter**, which switches the model to the **untrusted** state.

Summary

KSS is a security verdict engine that allows you to model a system as a set of security domains, describe interactions between these domains and associate rules (or policies) with the interactions. KSS SDK contains a rich set of policies that can be immediately implemented in the customer's solutions: type enforcement, role-based access control, temporal logic dialects, object capabilities, etc. If the provided set of security policies is not enough, new custom security models can be easily introduced with KSS SDK.

Kaspersky Security System

Read more on os.kaspersky.com

www.kaspersky.com