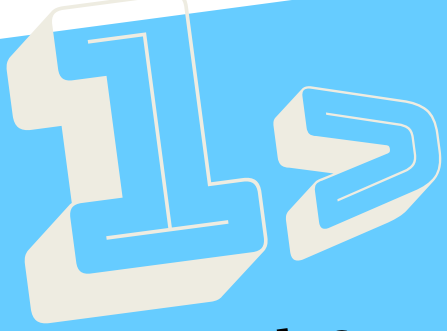
BULLET-PROOFING

YOUR SAP ABAP APPLICATIONS

A comprehensive guide
to enforce **security**
in your ABAP developments

kiuwan

CONTENTS>



Perspective

SAP is the 3rd largest software company in the world. It was founded in 1972 in Germany by 5 IBM engineers, who were already working on an enterprise-wide business system that IBM decided to cancel. Currently based in Walldorf, Germany, it has grown to be a global company serving 320,000 customers in 190 countries in all 5 continents *(Source: SAP Global Corporate Affairs, July 20, 2016)*.

They launched their first commercial product in 1973, the seed that became the first Enterprise Resource Planning (ERP) system. In the mid 1990s they moved from the mainframe to a client/server architecture and the explosion began. ERP systems provided an integrated view of core business process such as purchasing, orders, payroll, financial, human resources, inventory management and other more specialized process for specific industries like oil & gas, engineering, automotive, retail, healthcare, telecommunications. This variety of functionality, along with the client/server architecture that allowed more affordable infrastructure, expanded the use of ERP systems to companies of all sizes and industries. And SAP was there first.

Today, according to several sources, SAP is still the undisputed leader with around 25% of the ERP market share, doubling the second contender. This market share has been repeated year over year since before 2000, taking into consideration that the ERP global market has grown at an average of 7% yearly you can imagine the footprint SAP ERP system have in the enterprise landscape:

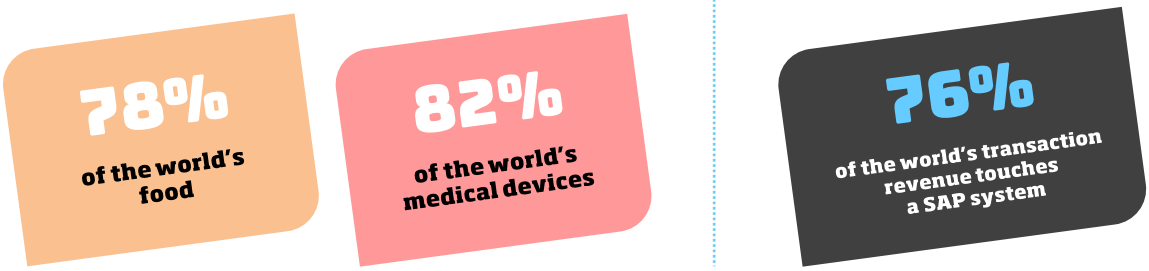
SAP customers include:

87% of the Forbes Global 2000 companies

98% of the most valued brands

(Source: SAP Global Corporate Affairs, July 20, 2016)

SAP customers produce:



(Source: SAP Global Corporate Affairs, July 20, 2016 | Oxford Economics/SAP analysis update 4/201)

ERP systems are complex beasts. The bigger the company implementing it, the more complex its ERP. Bear in mind that when we talk about global companies, they have to support all kinds of business processes across several company departments and geographies. Can an ERP solution fit all companies? Hardly. That is why customization and extensibility of the ERP platform of choice is a must for any company. SAP is no exception, and in fact it is one of the aspects of any SAP installation and deployment. Every out-of-the-box SAP module (more than 80) can be customized and extended ad infinitum.

SAP provides 2 ways to customize and extend its functionality:

- **Parameterization/Configuration**: To change the behavior of default functionality on different modules.
- **ABAP/WebDynPro custom code**: To develop new functionality to match specific business needs.

It is rare (very rare indeed!) to find a SAP installation that hasn't been customized and extended. To give you an idea of the customization efforts, one of our enterprise customers has more than 5 million lines of customized ABAP code running in their SAP environment. All this code has been developed by internal and external teams and has become a core element of the success of the company's operations. But what is ABAP? What can you do with it? Why is it key for the success of a SAP implementation?

The ABAP programming language

ABAP stands for Advanced Business Application Programming, originally it stood for Allgemeiner Berichts-Aufbereitungs-Prozessor and it was just a general report creation processor for the first versions of SAP. Today, it is a high-level general purpose programming language to extend the functionality of SAP systems. In fact in 2001 all but the most basic low-level functions of SAP were written in ABAP, including:

SAP_BASIS is the required technical base layer which is required in every ABAP system.
SAP_ABA contains functionalities which is required for all kinds of business applications, like business partner and address management.
SAP_UI provides the functionality to create SAP UI5 applications.
BBPCRM is an example for a business application, in this case the CRM application
SAP ABAP is a ERP programming language itself.

When the language evolved as part of the client/server architecture introduced with SAP R/3 it offered an abstraction between the business applications, the operating system and the database, ensuring applications don't depend directly upon a specific server or database and can be easily ported. This is fundamental to let developers focus on business functionality rather than in the intricacies and technicalities of the system. At the same time, it makes migrating between different versions of the underlying SAP system and infrastructure much easier and ensures business continuity.

ABAP programs execute in SAP environments as transactions. The normal way of executing ABAP code is by providing a transaction code (for example, VA01 is the transaction code for "Create Sales Order"). Transactions can be called via system-defined or user-specific, role-based menus. They can also be started by entering the transaction code directly into a command field, which is present in every SAP screen. Transactions can also be invoked programmatically by means of the ABAP statements CALL TRANSACTION and LEAVE TO TRANSACTION. The general notion of a transaction is called a Logical Unit of Work (LUW) in SAP terminology.

As in other programming languages, an ABAP program is either an executable unit or a library, which provides reusable code to other programs and is not independently executable.

ABAP distinguishes two types of executable programs:

Reports

Module pools

Reports follow a relatively simple programming model whereby a user optionally enters a set of parameters (e.g., a selection over a subSET of data) and the program then uses the input parameters to produce a report in the form of an interactive list. The term "report" can be somewhat misleading in that reports can also be designed to modify data; the reason why these programs are called reports is the "list-oriented" nature of the output they produce.

Module pools define more complex patterns of user interaction using a collection of screens. The term “screen” refers to the actual, physical image that the user sees. Each screen also has a "flow logic", which refers to the ABAP code implicitly invoked by the screens, which is divided into a "PBO" (Process Before Output) and "PAI" (Process After Input) section. In SAP documentation the term “dynpro” (dynamic program) refers to the combination of the screen and its flow logic.

The non-executable program types are:

- **INCLUDE modules**. They get included at generation time into the calling unit; it is often used to subdivide large programs.
- **Subroutine pools**. Contains ABAP subroutines (blocks of code enclosed by FORM/ENDFORM statements and invoked with PERFORM).
- **Function groups**. Libraries of self-contained function modules (enclosed by FUNCTION/ENDFUNCTION and invoked with CALL FUNCTION).
- **Object classes**. Similar to Java classes, they define a set of methods and attributes
- **Interfaces**. Similar to Java interface, they contain "empty" method definitions, for which any class implementing the interface must provide explicit code.
- **Type pools**. Define collections of data types and constants.

ABAP programs are composed of individual sentences (statements). The first word in a statement is called an ABAP keyword. Each statement ends with a period. Words must always be separated by at least one space. Statements can be indented as you wish. With keywords, additions and operands, the ABAP runtime system does not differentiate between upper and lowercase.

Statements can extend beyond one line. You can have several statements in a single line (though to ensure maintainability this is not recommended). Lines that begin with asterisk * in the first column are recognized as comment lines by the ABAP runtime system and are ignored. Double quotations marks (") indicate that the remainder of a line is a comment.

What does SAP systems offer in the security front?

According to some analysts, SAP systems security is not up to the level of criticality of the information they manage. Not only financial and human resources information but also engineering and new products' development information are at stake.

So are SAP systems insecure? How is that possible? Well, not everything is bad news, there's been a significant reduction on the reported and fixed security issues on SAP systems since a peak that took place in 2010. SAP bundles several features to handle security:

- Roles and authorization (explicit model. More about this later)
- Separation of duties (SoD)
- Separation of layers (3-tier system landscapes)
- Separation of clients (MANDT field in tables)
- Secure systems and SAP services configuration
- Network security infrastructure
- Cryptography (cypher, MAC, hash digest, digital signature…)
- Password policies
- Patches and transports management
- Identity management and single sign-on

Wait, with all this in place how can SAP systems still be insecure? It depends on how well or bad you use these features. For example, I assume you will never have in your system a default user SAP*, with default password and SAP_ALL privileges. Duh! But this still happens. A bigger concern, and more difficult to control, is how your ABAP customized applications use these features. This is a whole new ball game.

Do you have to worry about security in ABAP?

Code, of any nature, is an enterprise asset.

In particular, given the criticality for any business running a SAP system, customized ABAP applications become delicate and sensitive systems. They are like vital organs in a human body, they are the heart, the liver, the lungs, the kidneys of the organization running them. Are they sufficiently protected? Can your organization survive a heart attack?

Besides possible environment (operating systems, network) misconfiguration issues, what else can threaten the vital organs of your organization? We have to look at the application level:

- SAP system configuration
- Customized ABAP programs
- SAP webhooks and customized interfaces

I hope you have reached the obvious answer to our question: Yes, you have to worry about security in your ABAP applications. And the sooner you worry about it, the better. Like in any software development life cycle, and ABAP applications are no exception, defects and security vulnerabilities can be introduced at any time. Most likely and very often this happens in the coding phase. Harness your ABAP application development life cycle and prevent many security vulnerabilities reaching your production environments and sleep better at night.

ABAP security is not a walk in the park

SAP environments are usually very heterogeneous and complex. Different functional modules, versions, application stacks, databases, operating systems, you name it. Securing all this from a configuration point of view is a challenge in itself. Our concern here is how customized ABAP applications add complexity and broaden the attack surface of SAP.

ABAP is a proprietary and complex language. It offers several programing paradigms:

- Programs/reports
- Subroutines (FORM)
- Functions
- Dynpros
- Classes and methods
- BSP (Like JSP in Java)
- Web dynpros

On top of this, the core of most applications need database access to critical information. ABAP has a standard, integrated and portable (database independent) SQL interface to access information: OpenSQL, although there are other alternatives.

The SAP integrated roles and authentication model require explicit checks to access resources from ABAP applications (AUTHORITY-CHECK). This means you have to secure the code in your applications. Not only that, you can expose remote RFC functions that require you to manage S_RFC authorization objects that are complex to maintain.

So as you can see the security of your ABAP application heavily relies in your ABAP developers. And what is your worst enemy in this security pandemonium? IGNORANCE.

Take your ABAP devs out of the dark

SAP knows this, that's why not long ago they have added "Security Notes" to the ABAP documentation, but these are just a summary. To really help developers prevent security vulnerabilities requires other sources. This eBook could be a good starting point.

Make your ABAP developers aware that, like in any business oriented language, the vulnerabilities that allow information leakage have a much bigger impact than the technical vulnerabilities. In fact, information leakage vulnerabilities are by far the first cause of security problems in SAP systems. To that end, they have to know it is their responsibility to explicitly code the necessary authorization checks (AUTHORITY-CHECK) to access any sensitive SAP resources (functions, transaction, data) using the appropriate authorization object.

Many ABAP developers are often unaware of the dangers other "classical" more technical vulnerabilities (SQL injection, command injection, path traversal) pose to their babies.

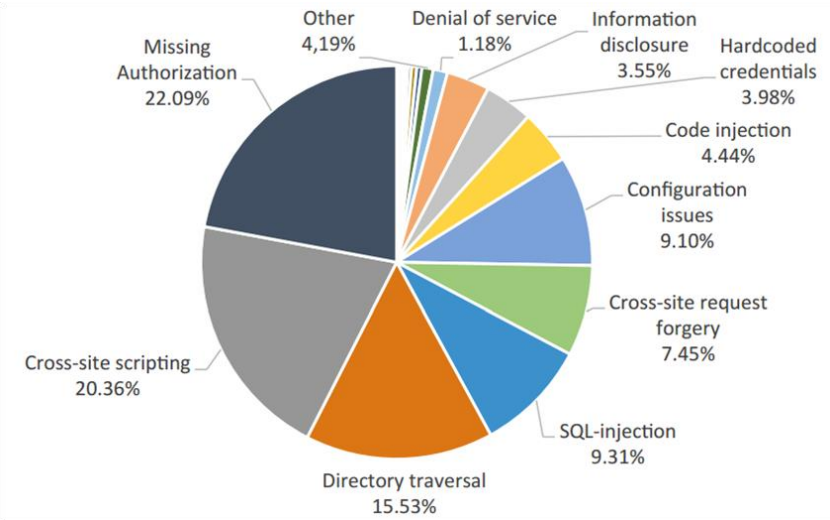
- What parts of the database access' API are exposed to SQL injection
- Regular expression evaluation in ABAP can be the backdoor for a DoS attack
- How to correctly code dynamic calls to OS commands

These things are foes for ABAP developers, simply because of plain ignorance. Not that it is their fault, SAP systems weren't so exposed to these when they started developing ABAP as they are today. However, don't let ignorance be an excuse.

A wise man and good friend once told me "Software is secure when it does what it is supposed to do… the right way. Nothing else". You can print that on t-shirts and give them to your ABAP developers.

Vulnerabilities distribution and attack surface

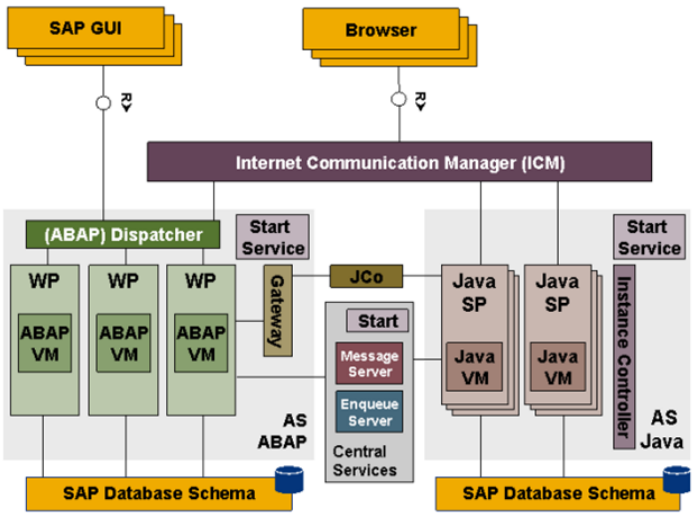
Here you have a figure with the distribution of reported security vulnerabilities, as described in the SAP Security Notes, affecting SAP's own ABAP code (the proprietary code implementing many SAP functions)



These results are aligned with what we said before, almost ¼ of the vulnerabilities are related to access control (missing, incorrect or badly implemented authorization). The second in line is cross-side scripting (XSS), relevant but not of much impact in this context as we will see later; path traversal is third, very easy to introduce but hardly known by the ABAP developer community; and then other classics like SQL injection, cross-side request forgery (CSRF) and code injection.

I think it is safe to extrapolate these results to custom ABAP code out there, so now you know what you are up against. To understand why these, and not others are the usual suspects we have to consider when coding ABAP applications, let's take a closer look at the SAP attack surface.

Source for upper graphic: ERPScan Research, "Analysis of 3000 vulnerabilities in SAP" (2014)



Here you have a classic SAP architecture diagram for R/3 and NetWeaver

In modern SAP implementations you can find other services like Fiori (richer user experience), Afaria (mobile device management), Business Objects, Mobile Platform, Hana (architecture beyond R/3), Business Warehouse or Lumira (data visualization). The flexibility (hence complexity) of a SAP system implies that there are many weak points that pose a threat to possible attacks. All these points put together are considered the attack surface.

For starters, you have the user entry points, GUI screens, web pages and mobile apps used to interact with the system. If you are serious about application security you should know that user input should always be treated as an “untrusted” source and your applications have to make sure the input can only be trusted depending on what the final destination of that input is going to be. Again it is the developer’s responsibility to check all user input.

Then you have protocols and APIs. The protocol for remote execution of ABAP programs and functions, RFC; web protocols HTTP/S, the protocols to communicate with databases, xDBC and SQL/MDX in Hana, if not used correctly can pose a serious threat to security. SAP exposes a good number of APIs and not all are always secure (ABAP programmers can even make direct calls to the SAP kernel) the aforementioned OpenSQL for example could be highly insecure if not used properly.

Another concern that adds up to the attack surface is the fact that SAP stores application data, configuration and system info and even the ABAP code itself in the same database . This opens the door (via SQL injection) to attacks that can install harmful code, change the security configuration to gain special privileges and, considering that SAP processes usually run with high OS privileges, attackers could potentially take full control of your system. No joke.

Bulletproofing one vulnerability at a time

Missing authorization checks

You can argue if it was a good decision or not, but it is what it is. SAP authorization model is explicit, so the responsibility of checking authorization to access resources is in the developer's hands. They have to do AUTHORITY-CHECK on authorization objects associated to roles before using those resources. There, language provides functions for commonly used resources/objects:

- AUTHORITY_CHECK_TCODE
- AUTHORITY_CHECK_DATASET
- CALL TRANSACTION ... WITH AUTHORITY_CHECK...

What resources should you consider sensitive? The obvious answer is all, but if you have to choose, go for: transactions, RFC functions or programs, system commands, files, SAP system tables and user tables with sensitive data (you should know which ones these are).

Even if there is an explicit check it could be weak or even useless if the sy-subrc return code is not treated appropriately or if DUMMY fields or '*' are used instead of explicit values for the protected resource.

Want to see what this looks like?

- Call to sensitive transaction without authorization check

```
1  " VIOLATION: No explicit authorization check
2  CALL TRANSACTION 'SE38' USING BDCDATA MODE 'N' MESSAGES INTO MESSTAB.
3  " Starts ABAP editor, where attacker may inject or alter code in SAP system
```

- RFC call without authorization check

```
1  * Caller
2  FUNCTION ZRUN IMPORTING filename TYPE string.
3    CALL FUNCTION 'ZRFC_FM' DESTINATION target_server
4      EXPORTING filename = filename.
5  ENDFUNCTION.
6
7  * RFC-enabled function: VIOLATION, no explicit authorization check
8  FUNCTION ZRFC_FM IMPORTING filename TYPE string.
9    DELETE DATASET filename.
10 ENDFUNCTION.
```

- Incorrect explicit check

```
17 " VIOLATIONS:
18 " P_GROUP should appear (e.g. as DUMMY),
19 " DEVCLASS should not be wildcard
20 " ACTVT should have a proper value
21 AUTHORITY-CHECK OBJECT 'S_DEVELOP'
22   ID 'DEVCLASS' FIELD '*'
23   ID 'OBJTYPE'  FIELD 'PROG'
24   ID 'OBJNAME'  FIELD lv_prog
25   ID 'ACTVT'    DUMMY.
26
27 IF sy-subrc = 0.
28   READ REPORT lv_prog INTO lt_code.
29 ENDIF.
```

SQL Injection

A classic security vulnerability since 1998, when it was documented for the first time in the PHRACK magazine. SAP as any other database application is not vulnerable to it.

The impact in SAP can be devastating. The attacker could directly write in business tables like ME21 or in system tables like REPOSRC or USR02, open back doors assigning SAP_ALL privileges to users in the UST04 table

The standard OpenSQL, however, is mostly secure. Using @param parameters in where clauses are secure:

```
SELECT col FROM T WHERE col2 = @var
```

is generally okay, the content of @var is treated by the ABAP runtime in a way that prevents any SQL injection even if it contains malicious input from an untrusted user input. But dynamic SQL like this

```
SELECT col FROM T WHERE (filter)
```

is not okay, in this case filter is usually constructed dynamically mixing column names and non-neutralized user input resulting in a SQL statement that can produce the injection.

Interestingly enough dynamic SQL with Open SQL is widely used because of its flexibility but it could be extremely dangerous if you don't take additional defensive measures to neutralize user input like:

- Positive validation patterns
- Use input dependent white-lists to build the dynamic part of the query
- Escape quote characters with the `quote()` and `quote_str()` from the `cl_abap_dyn_prg` class

With the above in mind try to always use Open SQL, it is portable, takes separation of clients (MANDT) into account and can use optimizations like table buffers. Other available APIs don't offer ANY protection against SQL injection, you are on your own if you use: the CL_SQL_STATEMENT/CL_SQLPREPARED_STATEMENT classes, DB_EXECUTE_SQL function, C_DB_EXECUTE/C_DB_FUNCTION kernel calls or the EXEC_SQL ... ENDEXEC construct.

Command injection

In many occasions, ABAP applications need to execute OS commands and they may get parameters to those commands from user input. This opens the door for attackers to try to inject commands in the input to be later executed by the SAP processes. If developers don't take the appropriate measures, the effects can be very harmful since SAP processes often have high OS privileges.

There is a SAP recommended way to invoke OS commands from ABAP programs. It involves a couple of things:

- Configure a white-list of allowed commands to execute with the SM69 SAP system transaction
- Use the function modules available in the SXPT function group like SXPG_CALL_SYSTEM and SXPG_COMMAND_EXECUTE
- This is not related to the injection itself, but to avoid information leaking always use the S_LOG_COM authorization object to authorize the calls

We can consider this method secure enough but not bulletproof. The reason for this is the negative validation the SXPT function modules do of some characters (like ;) included in the additional parameters to the calls that can be dangerous when interpreted by the shell. Negative validations are dangerous because attackers can "hide" these characters if our implementation does not normalize the input.

But we are not done yet! There are other APIs in ABAP that allow executing OS commands which can be potentially dangerous if the parameters are not neutralized: CALL 'SYSTEM', CALL 'ThWpInfo', OPEN DATASET ... FILTER, RFC_REMOTE_*, CL_GUI_FRONTEND_SERVICES->EXECUTE or TH_GREP. You should always avoid code like this:

```
3  DATA lv_cmd(255) TYPE c.
4  PARAMETERS pv_count TYPE i DEFAULT 5,
5  PARAMETERS pv_target TYPE c LENGTH 80 LOWER CASE.
6
7  lv_cmd = |ping -c{ pv_count } { pv_target }|.
8  CALL 'SYSTEM' ID 'COMMAND' FIELD lv_cmd ID 'TAB' FIELD lt_result.
```

Other types of injection

The flexibility of the ABAP language itself make possible other 'injection' attacks like:

- **Code injection**: Dynamic code sentences, like INSERT REPORT, GENERATE SUBROUTINE POOL or GENERATE REPORT / GENERATE DYNPRO can be a threat.
- **Dynamic language constructions**: ASSIGN (var) TO <fs>, SUBMIT (var), CALL FUNCTION var or CALL (class)=>method... can be "poisoned" with untrusted user input.
- **Path traversal**: Remember that this is the 3rd in most frequent vulnerability ranking. This is often not prevented properly despite SAP offering a relevant prevention mechanism (file logical naming).

Backdoors

A software backdoor is a "hidden feature" designed to avoid security controls. They are usually introduced by developers, but some language and operation systems have them to make live easier in certain circumstances.

Most times they are put in the code intentionally and with no malicious purpose, as I said to make the lives of developers easier in the development phase. The problem is when you forget to deactivate them before deploying them in production.

Others may be intentional and malicious and are introduced in the code by developers who want to exploit them once the system is rolled out. These ones are very difficult to detect before they are exploited with static analysis, because the offenders cover their tracks more or less carefully. However, there are some traces that can help us narrow down 'where to look':

- General queries to sensitive tables
- Direct modifications to tables such as REPOSRC or USR02
- “Stealth” techniques: #EC directives to change behavior, dynamic code, hidden OK codes.
- No standard authorization logic.

It is recommended that any automatic analysis is complemented with peer code reviews to find these and others.

Others are part of the ABAP language itself! And are often used by developers unintentionally out of ignorance. Known examples are:

- **OIUH_SUBMIT_UNIX_CALL**: Allows to upload script code to a remote server and execute it.
- **RS_FUNCTIONMODULE_INSERT**: It was originally created to allow execute remote code skipping over the testing system, hence breaking the separation of environments paradigm. It even includes a parameter to deactivate the authorization check.
- **RKC_FUNCTION_INTERFACE_GEN**: Like the previous one, was designed to skip over the testing system. In this case it allows to create a report with arbitrary ABAP code. Along with HR99B_PARALLEL_REPORT_RUN it can be executed remotely.

There are others you have to be careful with too, like RSRV, RS_FUNCTIONMODULE_INSERT or RKC_FUNCTION_INTERFACE_GEN.

Finally, an example of an usually unintentional back door is a misuse of OK codes in Dynpro programs. OK codes in Dynpro execute a function associated to an UI element event caught by a defined event listener (LINE-SELECTION, USER-COMMAND). For example, when a user presses a button in a SAPgui screen. It is more common than not that buttons are displayed or enabled depending on the state or the role of the user. However, SAPgui can expose a field for the user to insert an OK code, and OK codes are usually fairly easy to guess, so an attacker can directly invoke the function associated with the event even without permission.

```
2  DATA: ok_code LIKE sy-ucomm.
3  AT USER-COMMAND.
4    ok_code = sy-ucomm.
5    CASE ok_code.
6      WHEN 'SAVE'.
7        " ... accessible even with a hidden SAVE button ..
8    ENDCASE.
```

Cross client access

We mentioned in the introduction to SAP systems that they implement ”separation of clients”, meaning that one single instance can hold business information about different organizations (MANDanT support) If you access the information with OpenSQL every query is filtered by the client assigned to the accessing user (sy-mandt). Unfortunately, there are ways to override this behavior being able to expose sensitive information of other clients to the wrong users.

- Using the CLIENT SPECIFIED modifier in the queries
- Using native or not managed SQL (ADBC)

```
2  REPORT ZDUMP_ALL_HR.
3    TABLES: pa0008.  " HR Master Record: Infotype 0008 (Basic Pay)
4    SELECT * FROM pa0008 CLIENT SPECIFIED.
5       " SAP Client, User Name, Personnel Number, Annual Salary
6       WRITE: / pa0008-mandt, pa0008-uname, pa0008-pernr, pa0008-ansal.
7    ENDSELECT.
```

The resulting report of the above code is not only the best way to expose sensitive information for all the organization in the SAP system, it is a great way to get the HR manager to have a heart attack.

The web

We can’t forget the web. For several years now, you can develop rich web interfaces to customize your applications and facilitate the access to your users who don’t need a heavy SAPgui application on their desktops or laptops to use ERP functionality. The promise of universal access for SAP applications, even from outside your firewall. But this is not free, web technologies are dangerous from the security standpoint. More so if they are exposed to the Internet, so there you have it, the eternal dilemma between good and evil reincarnated: flexibility vs. security.

SAP offers a wide variety of web technologies to customize user experience for the Web: BSP, WebDynpro, WebGUI... Besides you can expose SAP functionality as web services (SOAP / REST) for other applications to consume.

Opening SAP applications to the web means that they automatically become potentially vulnerable to the top 10 vulnerabilities described by the Open Web Application Security Project, better known by its acronym: OWASP. Those will include cross-site scripting (XSS), cross-site request forgery (CSRF), server-side request forgery (SSRF), open redirect, etc.

The following example shows a BSP vulnerable to the classic cross-site scripting attack. It is using non-neutralized untrusted user input, so a savvy attacker could execute arbitrary Javascript code in the user's browser.

```
<%@page language="abap"%>
<%@extension name="htmlb" prefix="htmlb"%>
<% prod_id = request->get_form_field( 'prod_id' ). %>
<table>
<%
select * up to 10 rows
from ekko into table it_ekko
where ekko-prod_id = prod_id.

loop at it_ekko into wa_ekko.
%>
  <tr>
    <!-- VIOLACION, XSS! -->
    <td><%=prod_id%></td>
    <td><%=wa_ekko-ebeln%></td>
  </tr>
<% endloop. %></table>
```

Cryptography

The correct use of cryptographic primitives (cipher, MAC, one-way hash or digital signatures) is essential to implement application security controls. But this is a double-edged sword because an incorrect use defeats its purpose.

Some incorrect uses may be detected using static analysis, like:

- Weak algorithms (hash MD5, DES cipher, ECB mode…)
- Bad seed to generate pseudo-random numbers in a security control context.
- Improper initialization (padding, initial vector)
- Hard coded cipher/decipher keys.

SAP doesn't offer a complete standard cryptographic primitives API, very often developers rely on third party APIs (not always 100% secure), system calls or even they implement their own implementation. A road to disaster.

9 Are you scared enough? Conclusion & summary

If by now you are not running scared around the office sharing this eBook with your ABAP developers or you are not directly checking right now if any of the above is hidden in your ABAP code, congratulations! You have everything under control and you can consider your ABAP applications bulletproof.

Otherwise, keep reading, valuable advise follows to reduce the risk you face with your SAP ABAP applications and your ABAP software development life cycle.

If I have to pick one single conclusion it would be:

> **PREVENTION**
> **is better than cure**

I'm sure you've heard this a thousand times, in this context it translates to "Find vulnerabilities as early as possible in your SDLC, so as little as possible reaches production to pose a real threat to your organization". But prevention is not only better, it is also cheaper! Fixing a vulnerability found in the early stage of the SDLC can be up to 100 times cheaper than fixing it when it is production. Not taking into account the loss and damages you can incur in case the vulnerability is exploited.

10 key takeaways

1

In general terms application security is one of the biggest challenges you can face in software engineering today.

2

SAP systems are complex per-se. The ABAP language adds flexibility to your SAP system but contributes and increases complexity at the same time.

3

SAP and ABAP security features are not enough to bulletproof your applications. You need to actively develop security into your applications and prevent vulnerabilities.

4

There is no prevention without knowledge and awareness. Make all your ABAP developers understand the risks they are facing.

5

Give them tools to prevent them and mitigate them. Train them on the most common vulnerabilities and what they mean technically from the coding standpoint. This eBook is a good start.

6

Define a clear, easy to use ABAP secure coding policy and share it across the organization and even with your external providers. Let them even contribute to it and close the client-provider loop.

7

Define abuse cases to test and challenge your applications and your SAP system.

8

There is no such a thing as a silver bullet when it comes to preventing security vulnerabilities. Use combined approaches: Static analysis, peer code reviews and dynamic penetration testing. None of these alone will help much.

9

Automate as much as you can. With an ABAP secure coding policy and well defined abuse cases, you can implement high levels of automation of static and dynamic analysis using the right tools.

10

Streamline every practice in a well established Software Development Life Cycle. Integrate with ALM tools if they are in place and make sense.

Comprehensive approach to ABAP application security

SDLC

Penetration testing

Automatic static analysis

kiuwan

Peer code reviews

ABAP secure application policy / Abuse test cases

Development team awareness and training

Hardening / secure configuration

Risk management

Kiuwan is the ultimate software analytics platform to help you prevent security vulnerabilities integrated with your SDLC.

More at kiuwan.com

About the authors

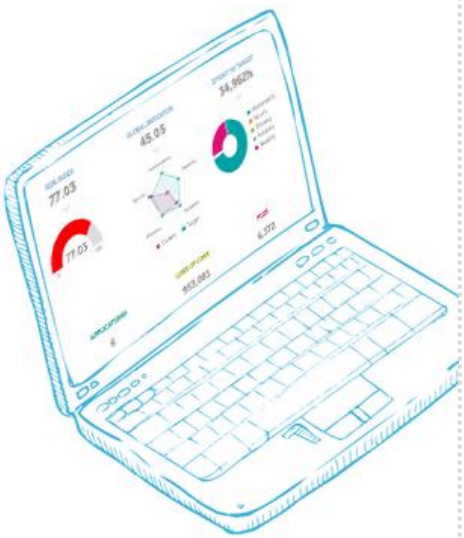
Luis Rodríguez

CTO at Optimyth/Kiuwan. Expert in Software Analytics and Application Security, he is the brains behind all Optimyth's analysis engines. International Speaker

Javier Salado

20 years experience in IT product/service management. He is an expert in ALM and security and quality assurance. International speaker

About Kiuwan



Kiuwan is Optimyth's software analytics solution in the cloud.

Over the last 10 years Optimyth has had a strong commitment with its customers to strengthen their application portfolios including their SAP ABAP applications that still run their businesses. Optimyth has heavily invested over the years to become leaders in static analysis for all kinds of technologies and gain customer's trust to protect and heal thair software assets.

Kiuwan's state of the art analysis technology is configurable, scalable and extensible. It allows customers to extract relevant software analytics from their application and make informed decisions based on them. As we like to say here at Optimyth "the truth is in the code". Let Kiuwan help your company find that truth and do something about it, no matter how ugly it may be.

THANK YOU!

Get in touch

Headquarters
2600 Lake Lucien Drive Suite 115 Maitland. FL 32751. USA

Call us: +1 9045 123 050 (USA)
Write us: contact@kiuwan.com
Partnership: partners@kiuwan.com

Try Kiuwan for free at kiuwan.com

kiuwan