2DMI20 - Software Security

Summary

The 3 pillars of software security

The 3 pillars of software security are:

1. Risk management
2. Touchpoints (between software development and software security)
3. Knowledge

The 7+1 pernicious kingdoms

The 7+1 pernicious kingdoms-model is used to abstractly categorize software issues into 8 so-called kingdoms. These are:

1. Input validation & representation issues
2. API abuse
3. Errors when using security tools/features
4. Parallelism & consistency issues
5. Error handling & output problems
6. Code quality
7. Encapsulation problems
8. Dependence on environment

F_x-measure

The F_x-measure is a generalization of the F-measure, which is also known as the F-score. The F-measure is a measure for *'accuracy'* which takes into account that the groups of positives and negatives may have vastly different sizes.

Note: refer to Canvas for more information about the relevancy of this topic.

The precision expresses completeness in a quantitative way. It is given by

$$precision = \frac{\#true\ negatives}{\#rejected}$$

The recall expresses soundness in a quantitative way. It is given by

$$recall = \frac{\#true\ positives}{\#accepted}$$

Finally, using these measures, the F_x-measure can be computed as

$$F_x = \frac{\left(1 + x^2\right) \cdot recall \cdot precision}{recall + x^2 \cdot precision}$$

The value of x in the F_x-measure determines the relative importance of recall and precision; more precisely, the measure is defined such that recall is x times as important as precision.

Memory layout

Text	Executable code
Data	Initialized variables
BSS Segment	Uninitialized variables
Heap	Dynamically allocated memory
Stack	Last-in, first-out memory

Buffer overflow

Attacker strategies

Code injection	Insert code into the overflown buffer. Cannot contain null bytes/string terminators.
Arc injection	Overwrite the return address to jump to another position in the program (e.g. skip password check).
Return-to-libc	Call functions already defined in memory, in particular those from `libc`. ~~Allows for what is called return-oriented programming, which has a Turing complete set of gadgets in `x86`.~~
Return-oriented programming	Similar to return-to-libc; however, this time, any arbitrary piece of data can be used to represent a function. For instance, by 'returning' to an address that points to a function, but off by one bit, one can get to data which represents a different set of instructions. This (at least in `x86`) allows for a Turing complete set of gadgets to be accessed/used.

Countermeasures

Fat pointers	Store length of memory with variable and check bounds at runtime.
Stack canary	Insert random value whose integrity is checked just before return address. Makes it much more difficult to modify the return address for an attacker, although it is vulnerable to situations where the stack content can be read by the attacker.
Data Execution Prevention ($W \oplus X$)	Mark certain regions of memory (i.e. data) as non-executable. Does not prevent return-to-libc/return-oriented programming, and does not work with just-in-time compilation (e.g. something like Jupyter notebooks).
Address Space Layout Randomization (ASLR)	Arrange memory positions of areas like heap, stack and code randomly. This makes it much more difficult for the attacker to correctly guess return addresses. Disadvantages include the vulnerability against `NOP`-sleds and leakage of ASLR addresses.

Heap and pointer issues

Some notes: NULL is a dedicated address that pointers cannot access. *Dereferencing* a pointer is another term for using that pointer.

NULL dereference	A pointer pointing to NULL is used.
Dangling pointer	A pointer points to an address that has been freed.
Use-after-free	A pointer is dereferenced after it has been freed.
Double-free	A pointer is freed after it has already been freed (and possibly re-allocated to a different program).
Memory leak	Allocated memory is never freed.

It is easy to confuse the lengths of string and arrays in C, which may lead to off-by-one errors. In particular, we note that `int char[10];` denotes a string of *'length 10'*, which ranges from `char[0]` to `char[9]`. However, since the last character in a C string is a termination symbol, `char[9]` is effectively always `\0`, which makes the actual string have length 9.

Integers are also sensitive to issues. In particular, they are vulnerable to overflow issues; that is, when an integer becomes too large, it can, for instance, roll over to restart from 0 or to go from the most positive integer to the most negative integer. Note that the exact behavior may depend on compiler optimizations.

Methods for finding security vulnerabilities

In general, there are three methods to find security vulnerabilities and/or bugs in software. These are:

1. Static code analysis
2. Fuzzing
3. Expert review (also known as penetration testing)

These methods have several advantages and disadvantages:

Method	Advantages	Disadvantages
Static code analysis	Automated	Can raise false alarms
	Cheap (from a computation point of view)	Does not find all bugs
	Does not require software execution	Does not find bugs which depend on runtime environment
	Covers corner cases	
	Finding source of bugs is straightforward	
Fuzzing	Automated	Computationally expensive
	Does not raise false alarms	Requires software execution
	Can find problems in deployed/production environment	Does not find all bugs
		Testing on production environment may be equivalent to an attack
	Can find errors in runtime environment	Does not cover the entire application
	Does not require access to source code	Does not indicate which code caused the issue
Expert review	No false alarms	Manual inspection
		Requires time (and is costly)
		Does not find all bugs

Since none of the methods finds all bugs, the methods can be used in a complementary manner.

Types of specifications

There are several types of specifications to formally specify (un)desirable properties that can be used in automated testing. We distinguish between:

- **History-based specification**, which considers the history of the system based on assertions which are being made over time.
- **State-based specification**, which specifies behavior based on system states or series of steps.
- **Transition-based specification**, which describes the system based on transitions between system states.
- **Functional specification**, in which the system is specified as a structure of mathematical functions. (This is quite similar to the concept of UCON/usage control from the course on Principles of Data Protection.)
- **Operational specification**, in which concepts like algebras or petri nets are used.

Safety properties deal with programs never reaching a bad state. Such properties are checked using e.g. assertions, a type system, pre- and postconditions or invariants. **Liveness properties** have to do with programs eventually reaching a bad state. Liveness properties are related to e.g. program termination and starvation freedom.

Static code analysis

Static code analysis uses many techniques to analyze source code. These include:
- Formalization and visualization of code dependencies
- Pattern checking (e.g. for code quality rules)
- Binary code instrumentation

It should be noted that many of these techniques overlap with compiler construction techniques.

Call & control flow graphs

A call graph contains a node for every function. Every edge in a call graph represents a function call.

A control flow graph contains a node for every straight-line code fragment (i.e. fragment without jumps). An edge in such a graph represents a jump. In the drawings we make for exercises, we tend to have the nodes contain a single statement (e.g. `x = y - 3`).

A static call graph represents every possible run of the program, including every call relationship present in the code (even if such a call would never occur in practice). Such a graph is (in general) an undecidable problem. Therefore, we usually limit ourselves to dynamic call graphs, which consider the parts of code which have been executed in a specific execution of the program.

Program verification

Program verification aims to establish correspondence between the program's actual behavior and its intended behavior. This can be done using specifications. Although a specification can involve a document of some sort, it can also be based on a comparison of the program to a different program. In this context, a program is correct if it complies with the specification.

However, due to constraints related to the halting problem, program verification generally approximates verification. This is acceptable, in particular because code written by humans is not arbitrarily complex.

Some approaches to program verification include:
1. Model checking, which explores all states and transitions in an automated fashion, but does not scale well.
2. Deductive verification, which generates proof obligations from the program and its specification. As a downside, it requires the user to understand why the system works correctly and convey this to the verification system.
3. Abstract interpretation, in which concrete sets of possible values are replaced by more abstract representations of that set (possibly of different size). In effect, abstract interpretation considers whether a strictly worse version satisfies the specification. (Properties of the program to consider can be simplified or split up if necessary.) A problem of abstract interpretation is that it still requires educated users.
   - As an example, abstract interpretation could check whether division by zero occurs by considering what values the denominator could have when a division occurs.

Data flow analysis

Data flow analysis aims to establish a set of facts at each point in the program. This can be used to analyze which program parts exchange data with each other, and also presents the dependencies resulting from this. Both forward and backward analysis variants can be used. There are several variants of data flow analysis. The classical variants include:
1. Reaching definitions analysis, which aims to find uninitialized variable uses;
2. Available expression analysis, which aims to avoid recomputing expressions;
3. Very busy expression analysis, which aims to reduce code size;

4. Live variables analysis, which aims to allocate registers efficiently.

Furthermore, there are several modern variants of data flow analysis:
5. Interval analysis, which checks for memory safety;
6. Taint analysis, which checks information flow (e.g. for data leaks or code injection);
7. Type-state analysis, which checks temporal safety properties (e.g. APIs of protocols/libraries);
8. Concurrency analysis, which considers e.g. data races and deadlocks

We will mainly focus on reaching definitions analysis and very busy expressions analysis.

Roughly, the idea of data flow analysis is to analyze the sets of values a variable can have. An important thing to note is that sets of possible values are monotonic, i.e. they can only grow or shrink. This implies that ***the set of all possible sets*** of values forms a lattice and ***is finite***.

Reaching definitions analysis

In reaching definitions analysis, the aim is to define, for the in- and output of every node[1] in the control flow graph, pairs of the following form:
`<x, 2> <y, 5>`
Such pairs express the names of variables and the points at which that variable's currently valid definition has been made. For example, `<x, 2>` means that the current value of the variable `x` could have been defined at the node numbered 2. More specifically, if a node would have `<x, 2>` and `<x, 4>` (and no more rules with `x`), that means that the current value of `x` was defined at either node 2 or node 4.

In a reaching definitions analysis, the IN and OUT sets (i.e. the sets of pairs at the in- and output of a node) for a node can only grow with more iterations of the algorithm. The algorithm for determining reaching definitions-pairs starts at the top of the control graph (i.e. at the start of the program) and ends at the bottom/exit.

Very busy expressions analysis

In very busy expressions analysis, the goal is to determine what are known as 'very busy' expressions at the exit of a point in the control flow graph. An expression is said to be **very busy** *'if, no matter what path is taken, the expression is used before any of the variables occurring in it are redefined'*. Such an analysis starts at the bottom of the control flow graph (i.e. at the end of the program), and works to the top.

A very busy expressions analysis can be used to determine which statements can be pre-computed before branching.

Differences between reaching definitions and very busy expressions analysis

The following is a tabular summary/comparison of the main differences between *reaching definitions analysis* and *very busy expressions analysis*.

Property	Reaching definitions analysis	Very busy expressions analysis
Working direction	Top-to-bottom Begin-to-end	Bottom-to-top End-to-begin
Initial table cell contents	Pairs of the form `<var_name, ?>` for the first cell Empty sets (for other cells)	One value for every expression. I.e. if `a-b` is computed at some point, every cell will contain a set containing `a-b`. Note: the IN-set of the final cell is always empty.

[1] Except for the input of the `entry` node and the output of the `exit` node.

Property	Reaching definitions analysis	Very busy expressions analysis
Update step	Mainly based on taking unions	Mainly based on taking intersections

Pointer analysis

Pointer analysis is an extension of data flow analysis, which aims to determine the set of objects to which a pointer can point. Such an analysis cannot be perfect due to the halting problem, and generally focuses on which pointers may alias a certain address. Pointer analysis is difficult because pointers can recur infinitely (think of e.g. a linked list, where we can point to the previous element of the next element infinitely many times). One way in which pointer analysis can work is by simplifying the program to a *'strictly worse'* program. For instance, a pointer to an array element can be simplified to a pointer to the entire array. Alternatively, fields of objects can be merged into a pointer to the entire object, or to a group of objects.

Fuzzing definitions

Fuzzing is the execution of a program[2] using inputs sampled semi-randomly from an input space which protrudes/exceeds the *expected* input space of the program. A *fuzzer* performs *fuzz testing*, which is the use of fuzzing to determine/test whether a program violates a *(security)* policy.
Some other relevant terms include the concept of a *fuzz campaign*, which indicates a specific execution of the fuzzer on a program, with a given configuration. The *fuzz configuration* consists of all variables/parameter values relevant to the execution of the fuzzer. These include:

- the program under test itself;
- possibly, one or multiple *seeds*, i.e. inputs that can be perturbed to produce random valid inputs for the testing. The seeds are collected in a so-called *seed pool*;
- possibly, variables which can be used to mutate the seeds in the seed pool.

Finally, for the execution of a fuzzer, a *bug oracle* is important; the bug oracle is an abstract representation of the 'thing' which determines whether the test passes or fails. The bug oracle effectively symbolizes the security policy being tested.

Generic fuzzing algorithm

Fuzzing algorithms can be expressed in a generic structure, which is given by the following algorithm:

```
C = PreProcess(C)
while (timeElapsed < timeLimit && Continue(C)) {
      conf = Schedule(C, timeElapsed, timeLimit)
      testCases = InputGen(conf)
      newBugs, executionInfo = InputEval(conf, testCases, oracle)
      C = ConfUpdate(C, conf, executionInfo)
      bugs.add(newBugs)
}
return bugs
```

Now, we will discuss the steps of this algorithm in somewhat more detail. First of all, we note that `C` is the set of all fuzz configurations, while `conf` is the fuzz configuration used in a given iteration of the fuzzer.

- `PreProcess` is a procedure which prepares the set of fuzz configurations/seeds. This can include aspects such as choosing the optimal seeds, but it can also involve adjusting the program under test or preparing some sort of model to select test cases.
- `Schedule` is a procedure which determines the fuzz configuration to be executed in this iteration of the algorithm. The goal of this procedure is to choose the 'optimal' configuration at this point in time. Choosing the optimal configuration can involve the time already used and the available time.

[2] More formally, we speak of a ***program under test***, or PUT. For ease of reading, we will simply call this 'program' in this summary.

- The main goal of the Schedule algorithm is to solve the *Fuzz Configuration Scheduling* (or FCS) problem, which entails the need to find a balance between looking for more favourable future decisions (**exploring**) or performing fuzzing to find actual outcomes (**exploiting**).

- `InputGen` is a procedure which uses the selected fuzz configuration to generate test cases to be executed. For example, this can be done on the basis of a model, by mutating seeds, by performing symbolic execution of the PUT or by mutating the PUT itself.
- `InputEval` is a procedure which uses the generated test cases, the fuzz configuration and the bug oracle to execute the test cases. This produces both a set of (*possibly* newly) discovered bugs and some information on the execution of the test cases, which can be used in the next step to optimally modify the set of fuzz configurations.
  - The bug oracle can perform several functions. For instance, it may indicate
    - Whether the program crashed
    - Whether there was a memory violation/error
    - Whether there was undefined behavior
    - Whether there was an input validation error
    - Whether the behavior was the same as another program. This check is known under the name *'semantic difference'*.
  - One problem that needs to be solved by the `InputEval` procedure is the *Fuzzer Taming problem*, which effectively deals with the ranking/prioritization ***of test cases*** (note: not of bugs) according to the uniqueness and severity of the found bugs.
- `ConfUpdate` is a procedure which modifies the set of fuzz configurations to ideally produce a minset: a minimal set of test cases that maximizes the coverage metric. Black-box fuzzers tend to skip this step, while grey-box and white-box fuzzers can use (for example) an evolutionary algorithm to optimize their fuzz configuration and seed pool. This procedure can use the current configuration, the set of all fuzz configurations and the execution info of this iteration as inputs, and produces an updated set of fuzz configurations as output.

Memory and type safety

A program execution is memory-safe if:
1. Pointers are only assigned through standard language features, and;
2. Pointers are only used to access memory allocated to those pointers.

A program is memory-safe if all of its executions are memory-safe.
A language is memory-safe if all possible programs written in that language are memory-safe.

There are two types of memory safety:
I. Spatial memory safety, which involves pointers reading/writing ***out of bounds***.
II. Temporal memory safety, which involves issues like use after free, double free and dangling pointers (i.e. related to pointers existing ***after*** free).

Type safety is strictly better than memory safety. A type-safety error occurs when memory is interpreted as a different type of variable than it was intended/assigned for. Abstractly speaking, type systems allow a programmer to *commit* to the set of values (i.e. *type*) that variables can hold and how they should be interpreted.
Type checking can be done at run-time (dynamic type checking) or at compile time (static type checking). The automatic detection of types is called type inference.

When complex types are used, the type system may implicitly re-interpret/convert types if these types (and the language) are compatible. Complex types may be equivalent, or one type may be a subtype of the other. When additional information is needed to compare types, this can be done in two ways:
- Nominal typing relies on the explicit definitions of types.
- Structural typing relies on the concrete structure of the type at compile time.

Web software security

Reasons for using database management systems (DBMS)

Database management systems can be used to help keep long-term data consistent and valid. They have four main properties, which help to guarantee validity of data even when errors and/or power failures occur:

Atomic	Statements are a single unit; they are fully executed or not at all.
Consistent	Database states are always valid; there are no ‘temporary invalid states’ (at least, they are not written to the database).
Isolation	Concurrently executed transactions behave as if done sequentially.
Durability	Once a transaction finishes, it persists, even in case of e.g. power failure.

SQL injection

For a general introduction to SQL injection as a strategy (without technical details), refer to my summary for 2IMS20. In this document, I will add a few notes on SQL syntax which may be useful on an exam.

The syntax `--` indicates the beginning of a comment. For instance, the line

```
SELECT * FROM table; -- this text gets ignored completely
```

ignores the text after the double dash. Similarly, we have syntax for comments which can span multiple lines. These can be written using the symbols `/*` and `*/`. An example could look like this:

```
SELECT * FROM table; /*
This line will not be executed.
*/
SELECT * from table; -- but this line will be executed.
```

Some other useful pieces of syntax, taken from the slides:

Query	**Effect**
`SELECT Balance FROM Accounts WHERE (UserID=‘Bob’ AND Password=‘5678’);`	Reads Bob’s balance
`INSERT INTO Accounts Values(‘Charlie’, ‘90AB’, 200); -- a comment`	Adds a new user Charlie, including data
`UPDATE Accounts SET Balance=‘200’ WHERE UserID=‘Alice’`	Overwrites Alice’s balance to 200
`DROP TABLE Accounts; /* Yet another, possibly multiline comment */`	Deletes the entire table named Accounts

Briefly re-summarized, a countermeasure for SQL injection is input validation, which can encompass whitelisting and/or sanitization, which includes the use prepared statements. Other countermeasures include data encryption and applying the principle of least privilege to table access.

HTTP ephemeral states

State in HTTP is ephemeral; that is, upon every request, some state/session (identifier) needs to be sent to the server. There are multiple ways to realise the storage of state on the client. These include the use of cookies or hidden fields in HTML documents. One disadvantage of using hidden fields is that all information is lost when the browser closes.

Cookies have multiple fields. These include its value, the domain it should be returned to, a path on that domain and an expiry date (which can also be end of session/upon closing the browser). It should be noted that cookies can be stolen.

CSRF

CSRF, or cross-site request forgery, is an attack in which the attacker attempts to make the victim perform an action which advantages the attacker on a site not under the control of the attacker. *(In other words, the attacker lets the client do what the attacker themselves are not allowed to do.)* In general, such an attack aims to make the victim perform a request to the site which ***modifies state***, such as a request which makes a payment/bank transfer. A site is vulnerable to an CSRF attack if it does not properly check whether a state-modifying request was *purposefully* initiated by the client from the legitimate website. Methods to prevent CSRF attacks include requiring a server-provided token (e.g. one included in legitimate forms) to be sent back with state-changing requests or (albeit less commonly) checking the `Referrer` header to see whether the submission came from a page on a trusted domain.

In general, it is often thought that the state-change mentioned in CSRF attacks require the user to be authenticated (and the attack aims to let an authenticated user perform the request on the attacker's behalf). However, different methods exist. In particular, an attacker can attempt to have the victim login to an account under the control of the attacker. Data entered into that account can then be accessed by the attacker as well.

JavaScript's same origin policy

The origin of a page is defined to be the combination of the *protocol, domain and port*. By the same origin policy, scripts included into page A can only access data in page B if they have the same origin.

XSS

XSS attacks aim to circumvent the same origin policy by 'tricking' the user's browser into believing the attacker's script comes from the legitimate site. Using this script, data such as keyboard input or cookie information can be exfiltrated to the attacker. There are several variants of XSS:

Persistent XSS/ Stored XSS	The attacker stores the script on the legitimate website, and then loads it from there. This usually involves a mechanism where the attacker can upload content to the site, such as a comment section or an attachment functionality.
Reflected XSS	The attacker makes the user send the script to the server, which then (wrongfully) sends it back to the user. Examples of this attack include a search functionality which echoes back the user's query.
Server-side XSS	In server-side XSS, the attacker makes the server access a website. This can be used to avoid firewalls, access resources from a trusted IP (e.g. localhost) or access backend systems.

Countermeasures to stored XSS include checking attachments to see whether they contain scripts, using multi-factor authentication for state-modifying requests and setting the `HttpOnly` flag on cookies.
Countermeasures to reflected XSS include avoiding the use of echoing and performing input validation to ensure the input does not contain any scripts.

Number theory and cryptography

Note: this section will only briefly mention the main points from the lectures, since there would be too many things to mention otherwise.

Extended Euclidean algorithm

The Extended Euclidean algorithm, or EEA, can be used to compute the greatest common divisor (`gcd`) of two integers a and b, and compute integers u and v such that $\gcd(a, b) = u \cdot a + v \cdot b$. u, v is called the EEA representation of $\gcd(a, b)$ with respect to a, b.

When $a > b$ and $\gcd(a, b) = 1$, we can compute $b^{-1} \mod a$ as the value of v produced by executing the Extended Euclidean algorithm on a and b.

Chinese remainder theorem

The Chinese remainder theorem (CRT) allows for finding a unique solution for $x \mod (p \cdot q)$, given a system of the following congruences:
$x = a \mod p$
$x = b \mod q$
This solution is given by $x = b \cdot u \cdot p + a \cdot v \cdot q \mod (p \cdot q)$, where u and v are obtained by applying the extended Euclidean algorithm to p and q (note: not to a and b). Also, note that u and v should be with the values (out of p and q) from which they were taken in the EEA. That is, if we were to take the EEA with q and p (i.e. because $q > p$), the roles of u and v would swap. Note that b is always multiplied with p and a is always multiplied with q.

Side-channel attacks

Side-channel attacks are based on extra information that is leaked from the implementation of the protocol, instead of from the design of the protocol/algorithm itself. Such attacks can in principle leak secret keys, and tend to be based on:
- Timing (e.g. time used to perform branches, memory cache)
- Error handling (e.g. Bleichenbacher oracle)
- Power analysis
- Electromagnetic leaks
- Sound

Timing attacks based on memory caching can partially be prevented by avoiding the use of data-dependent table lookups. In general, however, any form of extra information *(which can include compiler-introduced optimizations)* should be suppressed. Another countermeasure for side-channel attacks is to disturb correlations between extra information and data in a way that does not destroy functionality. A general way of doing this would be to insert some form of randomness into (the steps of) the computation.

RSA

The private and public key are related according to the relation

$$d = e^{-1} \mod \varphi(n)$$

where $\varphi(n) = p \cdot q$. Encryption occurs by raising the message to the power e, i.e. $c = m^e \mod n$. Decryption occurs by raising to the power d, i.e. $m = c^d \mod n$. Note that signing[3] is effectively the same as decrypting, while verifying the signature is effectively the same as encrypting.

RSA with CRT

The computation of RSA signatures can be sped up by first computing the following representation of the private key:

[3] To prevent some types of attacks, however, we generally do not sign the message itself but a hash of the message.

$$dp = d \mod \varphi(p)$$
$$dq = d \mod \varphi(q)$$

Then, the signature can be computed by first computing two 'parts' of it, which are known as sp and sq:

$$sp = m^{dp} \mod p$$
$$sq = m^{dq} \mod q$$

In general, we replace the message m with its hash $h(m)$. This gives the system of congruences

$$s = sp \mod n$$
$$s = sq \mod n$$

which can then be solved for s by applying the Chinese remainder theorem. This gives

$$s = sq \cdot u \cdot p + sp \cdot v \cdot q \mod n$$

As a reminder to myself: it seems that the use of Euler's φ-function is only necessary when computing d, dp and dq (and not when computing signatures or their components s, sp and sq).

Power analysis

Power analysis is the use of data on power consumption (e.g. in the square-and-multiply algorithm) to determine secret information (e.g. private keys).

Fault injection attacks

In a fault injection attack, the attacker deliberately makes a device (e.g. a smartcard) fail a particular computation in order to have it leak information. The RSA with CRT method described above is vulnerable to such an attack: if either sp or sq is computed incorrectly, while the other one remains correct, the difference between the hash and the resulting signature raised to the power e is divisible by one of the two factors p or q. In other words, we have that $\gcd\left(n, (s')^e - h(m)\right) \in \{p, q\}$. After obtaining one factor of n, the second one can easily be obtained by dividing n by the obtained factor.

Some countermeasures against attacks like this one include the following:

- Blind the value of d
- Blind the modulus n
- Blind some of the group elements (e.g. hashes or signatures)
- Verify the result of the computation before releasing it

Padding oracle

A padding oracle attack is based on issues related to error handling and outputs. The rough idea behind such an attack is that implementations of cryptographic protocols tend to require that the message is padded (i.e. zeros are prepended/appended to make the message fit in the size supported by the cryptographic system). When the server implicitly or explicitly indicates whether the padding is valid, this can be used to reconstruct the message by the attacker. Generally, the guessing can be done one byte at a time, which means that the attacker can do this significantly faster than brute-force.

Schnorr signatures

Schnorr signatures are a signature scheme based on discrete logarithms. The key generation is based on a Diffie-Hellman scheme, and works as follows:

1. Draw a random $a \in \mathbb{Z}_q$, which is output as the secret key sk
2. As the public key, output $pk = g^{sk} \mod p = g^a \mod p$

The signing scheme works as follows:

1. Draw a random $k \in \mathbb{Z}_q$
2. Compute $s_1 = H\left(m || g^k \mod p\right)$

3. Compute $s_2 = k - a * s_1 \mod q$
4. Output $s = (s_1, s_2)$

Verification of the signature occurs by checking whether the following equation holds:
$s_1 = H\left(m \,||\, g^{s_2} \cdot (pk)^{s_1} \mod p\right)$

The Schnorr signature scheme is vulnerable to attacks based on randomness; if two signatures use the same value of k, then, by rewriting the equation for s_2 to have k on one side, we have two equations for k with two unknowns: k and a, which is the secret key.
In my experience, the main things to remember about this scheme are the following; the rest of the scheme should be reconstructable from that:

- The equation for s_2;
- The fact that equations which involve g are taken $\mod p$, while equations which do not involve g are taken $\mod q$.
- Possibly, the fact that $p = 2q + 1$.
- Possibly, the fact that the hash is taken of the message concatenated with something equivalent to $g^k \pmod p$.

Attacks on randomness do occur in practice: in particular, the NSA managed to have the (later found to be backdoored) dual EC pseudorandom number generator standardized.