**1 Any source code anaylizing tool**

**Sonar**

Source code analysis tools, also referred to as Static Application Security Testing (SAST) Tools, are designed to analyze source code or compiled versions of code to help find security flaws. Some tools are starting to move into the IDE.

SonarQube is an open-source platform developed by SonarSource for continuous inspection of code quality. Sonar does static code analysis, which provides a detailed report of bugs, code smells, vulnerabilities, code duplications. Code coverage report using jacoco plugin

SonarQube is a web-based open source platform used to measure and analyse the source code quality.

mvnw clean

deletes target folder

mvnw package sonar:sonar

mvnw package sonar:sonar -Dsonar.login="admin" -Dsonar.password="sachin"

mvnw sonar:sonar -Dsonar.jdbc.url=jdbc:h2:tcp://ipaddr:9092/sonar -Dsonar.host.url=http://ipaddr:9000

mvnw sonar:sonar -Dsonar.jdbc.url=jdbc:h2:tcp://localhost:9092/sonar -Dsonar.host.url=http://localhost:9000 -Dsonar.login="admin" -Dsonar.password="sachin"

for code ccoverage report in sonar

mvnw jacoco:prepare-agent

mvnw jacoco:report

**2 how have you achieved abstraction in your project**

Yes

**3 Have you implemented oops concept and where?**

Yes Inheritance, Abstraction , overloading , overriding , encapsulation in entity

**4 what are the exception encounter during project**

1 Null pointer, ArrayBound of Exception,Number Format,

2 SQLGrammer Excpetion, ClassNotFundException,NODefFoundException , JsonParserException

**5 Git commands few**

**6 water fall model r agile**

Agile

**7 sprint duration and team size**

**4 weeks**

8 members

**8 advantage agile over water flow model**

Waterfall suits projects with well-defined requirements where no changes are expected. Agile looks best where there is a higher chance of frequent requirement changes. Waterfall is easy to manage and a sequential approach. Agile is very flexible and allows to make changes in any phase.

If the project timeline is fixed and can not be moved, Waterfall will offer a more predictable outcome. If you need to get the project delivered in a short amount of time, Agile is the appropriate choice here where action and getting things built is more important than documentation and process.Developer and tester can be occupied busy all the time in between any changes can be measured it will be independent modules.

**9 what all meeting(ceremony) will have for agile flow**

Daily Scurm, Spring reviews, retro section, refinement session, BI planning

**10 Explain Singleton pattern in multi thread environment why double check lock is required**

public class DclSingleton {

private static volatile DclSingleton instance;

public static DclSingleton getInstance() {

if (instance == null) {

synchronized (DclSingleton .class) {

if (instance == null) {

instance = new DclSingleton();

}

}

}

return instance;

}

// private constructor and other methods...

}

**11 to develop synchrouse and asynchronus service how to do it?**

**RestTemplate ,**

Client client = ClientBuilder.newClient();

WebTarget target = client.target(url);

Invoker.Builder builder = target.request();

AsyncInvoker asyncInvoker = builder.async();

**12 how to hold when getting huge amoutn of data is been gt by DB side to applcaition**

**13 login Framwork**

**14 Audit in application ?how many request came ?**

Using AOP and AUDIT can trace out request and time out count who , when and where,

ELK also used to keep track Request count

**15 coding standards in application**

Maintained using Sonar

**16 CICD for deployment (Process) -**

Check-in code - Devops comes -> code build in Jenkins(Once all test case and SAST and code vulnerability checked ) and pushed to servers- deployed.

**17 idea to improve application in current company**

Yes handling SMS gateway all huge consumer and its logs in ELK of keeping track

18 sonar cube scan configuration and how many call tickets (Major , minor , blocker)

**19 GIT and SVN diff**

**20 mvn build cmd**

Mvn clean

Mvn verify

Mvn test

mvn package

Mvn install

Mvn deploy

validate

generate-sources

process-sources

generate-resources

process-resources

compile

**21 does mvn build will delete something**

he Maven Clean Plugin will delete the target directory by default. You may configure it to delete additional directories and files.

**26 diff between java script and jquery**

**27 Product backlog**

A **sprint backlog** is the set of items that a cross-functional product team selects from its product **backlog** to work on during the upcoming **sprint**. Typically the team will agree on these items during its **sprint** planning session.

A **product backlog** is a prioritized list of work for the development team that is derived from the roadmap and its requirements. The most important items are shown at the top of the **product backlog** so the team knows what to deliver first.

**The Product Owner** (PO) “owns” the product backlog on behalf of the stakeholders, and is primarily responsible for creating it.

**28wt will happen if we don’t follow solid principles**

**29 List<Employee> , convert the list into Map<name List<EMployee>> usign java 8 streams group by fucnton of Streams**

30 customer is buying few products at some shop shop will givesome discount adn discoutn was sowmthin like [1,2,3,4]

fr 1st item no discount fr 2nd iteam discut would be current price minus prev price and for thirs and 4th item will discountof

current prive minus least price 1+1+2+4=8

**31 looping data 100 records excpeion in loop but still need all success reored in list**

**Continue nxtrow**

**32 making aynchous service into synchrnous service**

**33 what are java script types**

**34 java script diff from java**

**35 how to access an external javascript file strored external not embedded**

**36 how to secure code meets standards?**

A secure code review is a specialized task involving manual and/or automated review of an application's source code in an attempt to identify security-related weaknesses (flaws) in the code.

**Validate input.** Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software [vulnerabilities](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions" \l "BB.Definitions-vulnerability). Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files .

**Heed compiler warnings**. Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code

**Architect and design for security policies.** Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.

**Keep it simple.** Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use.

**Default deny.** Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted.

**Adhere to the principle of least privilege.** Every process should execute with the the least set of privileges necessary to complete the job. Any elevated permission should only be accessed for the least amount of time required to complete the privileged task. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges [Saltzer 74, Saltzer 75].

**Sanitize data sent to other systems.** Sanitize all data passed to complex subsystems [[C STR02-A](https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems)] such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.

**Practice defense in depth.** Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a [security flaw](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions" \l "BB.Definitions-securityflaw) from becoming an exploitable vulnerability and/or limit the consequences of a successful [exploit](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions" \l "BB.Definitions-exploit). For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment [Seacord 05].

**Use effective quality assurance techniques.** Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions [Seacord 05].

**Adopt a secure coding standard.** Develop and/or apply a secure coding standard for your target development language and platform.

**37 top 10 owasp security issues?**

The Open Web Application Security Project (OWASP) is a non-profit organization dedicated to providing unbiased, practical information about application security.

OWASP Top 10 Web Application Security Risks, and offers solutions and best practices to prevent or remediate them.

**1. Injection**

Injection flaws, such as SQL injection, LDAP injection, and CRLF injection, occur when an attacker sends untrusted data to an interpreter that is executed as a command without proper authorization.

\* Application security testing can easily detect injection flaws. Developers should use parameterized queries when coding to prevent injection flaws.

**2. Broken Authentication and Session Management**

Incorrectly configured user and session authentication could allow attackers to compromise passwords, keys, or session tokens, or take control of users’ accounts to assume their identities.

\* Multi-factor authentication, such as FIDO or dedicated apps, reduces the risk of compromised accounts.

**3. Sensitive Data Exposure**

Applications and APIs that don’t properly protect sensitive data such as financial data, usernames and passwords, or health information, could enable attackers to access such information to commit fraud or steal identities.

\* Encryption of data at rest and in transit can help you comply with data protection regulations.

**4. XML External Entity**

Poorly configured XML processors evaluate external entity references within XML documents. Attackers can use external entities for attacks including remote code execution, and to disclose internal files and SMB file shares.

\* Static application security testing (SAST) can discover this issue by inspecting dependencies and configuration.

**5. Broken Access Control**

Improperly configured or missing restrictions on authenticated users allow them to access unauthorized functionality or data, such as accessing other users’ accounts, viewing sensitive documents, and modifying data and access rights.

\* Penetration testing is essential for detecting non-functional access controls; other testing methods only detect where access controls are missing.

**6. Security Misconfiguration**

This risk refers to improper implementation of controls intended to keep application data safe, such as misconfiguration of security headers, error messages containing sensitive information (information leakage), and not patching or upgrading systems, frameworks, and components.

\* Dynamic application security testing (DAST) can detect misconfigurations, such as leaky APIs.

**7. Cross-Site Scripting**

Cross-site scripting (XSS) flaws give attackers the capability to inject client-side scripts into the application, for example, to redirect users to malicious websites.

\* Developer training complements security testing to help programmers prevent cross-site scripting with best coding best practices, such as encoding data and input validation.

**8. Insecure deserialization**

Insecure deserialization flaws can enable an attacker to execute code in the application remotely, tamper or delete serialized (written to disk) objects, conduct injection attacks, and elevate privileges.

\* Application security tools can detect deserialization flaws but penetration testing is frequently needed to validate the problem.

**9. Using Components With Known Vulnerabilities**

Developers frequently don’t know which open source and third-party components are in their applications, making it difficult to update components when new vulnerabilities are discovered. Attackers can exploit an insecure component to take over the server or steal sensitive data.

\* Software composition analysis conducted at the same time as static analysis can identify insecure versions of components.

**10. Insufficient Logging and Monitoring**

The time to detect a breach is frequently measured in weeks or months. Insufficient logging and ineffective integration with security incident response systems allow attackers to pivot to other systems and maintain persistent threats.

**11 how to find security issues of an application?**

**Authentication**: Verifying that a person is (or at least appears to be) a specific user, since he/she has correctly provided their security credentials (password, answers to security questions, fingerprint scan, etc.).

**Authorization**: Confirming that a particular user has access to a specific resource or is granted permission to perform a particular action.

Stated another way, authentication is knowing who an entity is, while authorization is knowing what a given entity can do. With this in mind, let’s get into the top 10 internet security issues.

**Mistake #1: Injection flaws**

Injection flaws result from a classic failure to filter untrusted input. It can happen when you pass unfiltered data to the SQL server (SQL injection), to the browser (XSS), to the LDAP server (LDAP injection), or anywhere else. The problem here is that the attacker can inject commands to these entities, resulting in loss of data and hijacking clients’ browsers.

Anything that your application receives from untrusted sources must be filtered, preferably according to a whitelist. You should almost never use a blacklist, as getting that right is very hard and usually easy to bypass. Antivirus software products typically provide stellar examples of failing blacklists. Pattern matching does not work.

Prevention: The good news is that protecting against injection is “simply” a matter of filtering your input properly and thinking about whether an input can be trusted. But the bad news is that all input needs to be properly filtered, unless it can unquestionably be trusted (but the saying “never say never” does come to mind here).

In a system with 1,000 inputs, for example, successfully filtering 999 of them is not sufficient, as this still leaves one field that can serve as the Achilles heal to bring down your system. And you might think that putting an SQL query result into another query is a good idea, as the database is trusted, but if the perimeter is not, the input comes indirectly from guys with malintent. This is called [Second Order SQL Injection](https://en.wikipedia.org/wiki/SQL_injection" \l "Second_Order_SQL_Injection) in case you’re interested.

Since filtering is pretty hard to do right (like crypto), what I usually advise is to rely on your framework’s filtering functions: they are proven to work and are thoroughly scrutinized. If you do not use frameworks, you really need to think hard about whether not using them really makes sense in your server security context. 99% of the time it does not.

**Mistake #2: Broken Authentication**

This is a collection of multiple problems that might occur during broken authentication, but they don’t all stem from the same root cause.

Assuming that anyone still wants to roll their own authentication code in 2014 (what are you thinking??), I advise against it. It is extremely hard to get right, and there are a myriad of possible pitfalls, just to mention a few:

The URL might contain the session id and leak it in the referer header to someone else.

The passwords might not be encrypted either in storage or transit.

The session ids might be predictable, thus gaining access is trivial.

Session fixation might be possible.

Session hijacking might be possible, timeouts not implemented right or using HTTP (no SSL security), etc…

Prevention: The most straightforward way to avoid this web security vulnerability is to use a framework. You might be able to implement this correctly, but the former is much easier. In case you do want to roll your own code, be extremely paranoid and educate yourself on what the pitfalls are. There are quite a few.

**Mistake #3: Cross Site Scripting (XSS)**

This is a fairly widespread input sanitization failure (essentially a special case of [common mistake #1](https://www.toptal.com/security/10-most-common-web-security-vulnerabilities" \l "CommonMistake_Injection)). An attacker gives your web application JavaScript tags on input. When this input is returned to the user unsanitized, the user’s browser will execute it. It can be as simple as crafting a link and persuading a user to click it, or it can be something much more sinister. On page load the script runs and, for example, can be used to post your cookies to the attacker.

Prevention: There’s a simple web security solution: don’t return HTML tags to the client. This has the added benefit of defending against HTML injection, a similar attack whereby the attacker injects plain HTML content (such as images or loud invisible flash players) – not high-impact but surely annoying (“please make it stop!”). Usually, the workaround is simply converting all [HTML entities](https://www.toptal.com/designers/htmlarrows/symbols/), so that <script> is returned as &lt;script&gt;. The other often employed method of sanitization is using regular expressions to strip away HTML tags using regular expressions on < and >, but this is dangerous as a lot of browsers will interpret severely broken HTML just fine. Better to convert all characters to their escaped counterparts.

**Mistake #4: Insecure Direct Object References**

This is a classic case of trusting user input and paying the price in a resulting security vulnerability. A direct object reference means that an internal object such as a file or database key is exposed to the user. The problem with this is that the attacker can provide this reference and, if authorization is either not enforced (or is broken), the attacker can access or do things that they should be precluded from.

For example, the code has a download.php module that reads and lets the user download files, using a CGI parameter to specify the file name (e.g., download.php?file=something.txt). Either by mistake or due to laziness, the developer omitted authorization from the code. The attacker can now use this to download any system files that the user running PHP has access to, like the application code itself or other data left lying around on the server, like backups. Uh-oh.

Another common vulnerability example is a password reset function that relies on user input to determine whose password we’re resetting. After clicking the valid URL, an attacker can just modify the username field in the URL to say something like “admin”.

Incidentally, both of these examples are things I myself have seen appearing often “in the wild”.

Prevention: Perform user authorization properly and consistently, and whitelist the choices. More often than not though, the whole problem can be avoided by storing data internally and not relying on it being passed from the client via CGI parameters. Session variables in most frameworks are well suited for this purpose.

**Mistake #5: Security misconfiguration**

In my experience, web servers and applications that have been misconfigured are way more common than those that have been configured properly. Perhaps this because there is no shortage of ways to screw up. Some examples:

Running the application with debug enabled in production.

Having directory listing enabled on the server, which leaks valuable information.

Running outdated software (think WordPress plugins, old PhpMyAdmin).

Having unnecessary services running on the machine.

Not changing default keys and passwords. (Happens way more frequently than you’d believe!)

Revealing error handling information to the attackers, such as stack traces.

Prevention: Have a good (preferably automated) “build and deploy” process, which can run tests on deploy. The poor man’s security misconfiguration solution is post-commit hooks, to prevent the code from going out with default passwords and/or development stuff built in.

**Mistake #6: Sensitive data exposure**

This web security vulnerability is about crypto and resource protection. Sensitive data should be encrypted at all times, including in transit and at rest. No exceptions. Credit card information and user passwords should never travel or be stored unencrypted, and passwords should always be hashed. Obviously the crypto/hashing algorithm must not be a weak one – when in doubt, web security standards recommend [AES (256 bits and up)](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) and [RSA (2048 bits and up)](https://en.wikipedia.org/wiki/RSA_numbers" \l "RSA-2048).

And while it goes without saying that session IDs and sensitive data should not be traveling in the URLs and sensitive cookies should have the secure flag on, this is very important and cannot be over-emphasized.

Prevention:

In transit: Use [HTTPS](https://en.wikipedia.org/wiki/HTTP_Secure) with a proper certificate and [PFS (Perfect Forward Secrecy)](https://en.wikipedia.org/wiki/Forward_secrecy" \l "Perfect_forward_secrecy). Do not accept anything over non-HTTPS connections. Have the secure flag on cookies.

In storage: This is harder. First and foremost, you need to lower your exposure. If you don’t need sensitive data, shred it. Data you don’t have can’t be [stolen](https://krebsonsecurity.com/2017/12/the-market-for-stolen-account-credentials/). Do not store credit card information ever, as you probably don’t want to have to deal with being [PCI compliant](https://www.pcisecuritystandards.org/security_standards/index.php). Sign up with a payment processor such as [Stripe](https://stripe.com/) or [Braintree](https://www.braintreepayments.com/). Second, if you have sensitive data that you actually do need, store it encrypted and make sure all passwords are hashed. For hashing, use of [bcrypt](http://bcrypt.sourceforge.net/) is recommended. If you don’t use bcrypt, educate yourself on [salting](https://en.wikipedia.org/wiki/Salt_(cryptography)) and [rainbow tables](https://en.wikipedia.org/wiki/Rainbow_table).

And at the risk of stating the obvious, do not store the encryption keys next to the protected data. That’s like storing your bike with a lock that has the key in it. Protect your backups with encryption and keep your keys very private. And of course, don’t lose the keys!

**Mistake #7: Missing function level access control**

This is simply an authorization failure. It means that when a function is called on the server, proper authorization was not performed. A lot of times, developers rely on the fact that the server side generated the UI and they think that the functionality that is not supplied by the server cannot be accessed by the client. It is not as simple as that, as an attacker can always forge requests to the “hidden” functionality and will not be deterred by the fact that the UI doesn’t make this functionality easily accessible. Imagine there’s an /admin panel, and the button is only present in the UI if the user is actually an admin. Nothing keeps an attacker from discovering this functionality and misusing it if authorization is missing.

Prevention: On the server side, authorization must always be done. Yes, always. No exceptions or vulnerabilities will result in serious problems.

**Mistake #8: Cross Site Request Forgery (CSRF)**

This is a nice example of a [confused deputy](https://en.wikipedia.org/wiki/Confused_deputy_problem) attack whereby the browser is fooled by some other party into misusing its authority. A 3rd party site, for example, can make the user’s browser misuse it’s authority to do something for the attacker.

In the case of CSRF, a 3rd party site issues requests to the target site (e.g., your bank) using your browser with your cookies / session. If you are logged in on one tab on your bank’s homepage, for example, and they are vulnerable to this attack, another tab can make your browser misuse its credentials on the attacker’s behalf, resulting in the confused deputy problem. The deputy is the browser that misuses its authority (session cookies) to do something the attacker instructs it to do.

Consider this example:

Attacker Alice wants to lighten target Todd’s wallet by transfering some of his money to her. Todd’s bank is vulnerable to CSRF. To send money, Todd has to access the following URL:

http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243

After this URL is opened, a success page is presented to Todd, and the transfer is done. Alice also knows, that Todd frequently visits a site under her control at blog.aliceisawesome.com, where she places the following snippet:

<img src=http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243 width=0 height=0 />

Upon visiting Alice’s website, Todd’s browser thinks that Alice links to an image, and automatically issues an HTTP GET request to fetch the picture, but this actually instructs Todd’s bank to transfer $1500 to Alice.

Incidentally, in addition to demonstrating the CSRF vulnerability, this example also demonstrates altering the server state with an idempotent HTTP GET request which is itself a serious vulnerability. HTTP GET requests must be [idempotent](https://en.wikipedia.org/wiki/Idempotence) (safe), meaning that they cannot alter the resource which is accessed. Never, ever, ever use idempotent methods to change the server state.

Fun fact: CSRF is also the method people used for cookie-stuffing in the past until affiliates got wiser.

Prevention: Store a secret token in a hidden form field which is inaccessible from the 3rd party site. You of course always have to verify this hidden field. Some sites ask for your password as well when modifying sensitive settings (like your password reminder email, for example), although I’d suspect this is there to prevent the misuse of your abandoned sessions (in an internet cafe for example).

**Mistake #9: Using components with known vulnerabilities**

The title says it all. I’d again classify this as more of a maintenance/deployment issue. Before incorporating new code, do some research, possibly some auditing. Using code that you got from a random person on [GitHub](https://github.com/) or some forum might be very convenient, but is not without risk of serious web security vulnerability.

I have seen many instances, for example, where sites got [owned](https://en.wikipedia.org/wiki/Owned) (i.e., where an outsider gains administrative access to a system), not because the programmers were stupid, but because a 3rd party software remained unpatched for years in production. This is happening all the time with WordPress plugins for example. If you think they will not find your hidden phpmyadmin installation, let me introduce you to dirbuster.

The lesson here is that software development does not end when the application is deployed. There has to be documentation, tests, and plans on how to maintain and keep it updated, especially if it contains 3rd party or open source components.

Prevention:

Exercise caution. Beyond obviously using caution when using such components, do not be a copy-paste coder. Carefully inspect the piece of code you are about to put into your software, as it might be broken beyond repair (or in some cases, intentionally malicious—web security attacks are sometimes unwittingly invited in this way).

Stay up-to-date. Make sure you are using the latest versions of everything that you trust, and have a plan to update them regularly. At least subscribe to a newsletter of new security vulnerabilities regarding the product.

**Mistake #10: Unvalidated redirects and forwards**

This is once again an input filtering issue. Suppose that the target site has a redirect.php module that takes a URL as a GET parameter. Manipulating the parameter can create a URL on targetsite.com that redirects the browser to malwareinstall.com. When the user sees the link, they will see targetsite.com/blahblahblah which the user thinks is trusted and is safe to click. Little do they know that this will actually transfer them onto a malware drop (or any other malicious) page. Alternatively, the attacker might redirect the browser to targetsite.com/deleteprofile?confirm=1.

It is worth mentioning, that stuffing unsanitized user-defined input into an HTTP header might lead to [header injection](https://en.wikipedia.org/wiki/HTTP_header_injection) which is pretty bad.

Prevention: Options include:

Don’t do redirects at all (they are seldom necessary).

Have a static list of valid locations to redirect to.

Whitelist the user-defined parameter, but this can be tricky.

**12 tool used for security issues**

security testing tool is SonarQube

In addition to exposing vulnerabilities, it is used to measure the source code quality of a web application. Some of the vulnerabilities exposed by SonarQube include:

Cross-site scripting

Denial of Service (DoS) attacks

HTTP response splitting

Memory corruption

SQL injection

Key highlights:

Detects tricky issues

DevOps integration

Set up analysis of pull requests

Supports quality tracking of both short-lived and long-lived code branches

Offers [Quality Gate](https://docs.sonarqube.org/7.4/user-guide/quality-gates/" \t "https://hackr.io/blog/_blank)

Visualize history of a project

**40 latching wrt multithreading**

CountDownLatch class is a synchronization aid which allows one or more thread to wait until the mandatory operations are performed by other threads.

**41 how to limit object creation on factory design pattern**

**42 type of exception faced in project and how to handle it**

**43 factory and abstract factoey diff**

**44 implement IOC using core java**

**45 caching in out application**

**46 inside bean class if creating new object Test t= new Test() how will t beahve what will be the scope**