TOPIC 6: WEB SECURITY

T6.1) Explaining web security

Definition of Web Security

1. Web application security is the process of securing confidential data stored online from unauthorized access and modification. This is accomplished by enforcing stringent policy measures. Security threats can compromise the data stored by an organization is hackers with malicious intentions try to gain access to sensitive information.
2. a set of procedures, practices, and technologies for assuring the reliable, predictable operation of web servers, web browsers, other programs that communicate with web servers, and the surrounding Internet infrastructure.

The aim of Web application security is to identify the following:

- Critical assets of the organization
- Genuine users who may access the data
- Level of access provided to each user
- Various vulnerabilities that may exist in the application
- Data criticality and risk analysis on data exposure
- Appropriate remediation measures

Web application security aims to address and fulfill the four conditions of security, also referred to as principles of security:

- Confidentiality: States that the sensitive data stored in the Web application should not be exposed under any circumstances.
- Integrity: States that the data contained in the Web application is consistent and is not modified by an unauthorized user.
- Availability: States that the Web application should be accessible to the genuine user within a specified period of time depending on the request.
- Nonrepudiation: States that the genuine user cannot deny modifying the data contained in the Web application and that the Web application can prove its identity to the genuine user.

The process of security analysis runs parallel with Web application development. The group of programmers and developers who are responsible for code development are also responsible for the execution of various strategies, post-risk analysis, mitigation and monitoring.

T6.2) Identifying web security issues

What are the different types of **website security issues, risks or threats**, and what can make your business and website an attractive or susceptible target? Many small businesses feel they do not represent a worthwhile target to attackers, but as you will read, this assumption is plain wrong. All online entities face a variety of security risks and threats that should be understood and assessed.

For all too many companies, it's not until *after* a breach has occurred that web security becomes a priority. An effective approach to IT security must, by definition, be proactive and defensive. Toward that end, this discussion is aimed at sparking a security mindset, hopefully injecting the reader with a healthy dose of paranoia.

In particular, this guide focuses on common and significant web security pitfalls to be aware of, including recommendations on how they can be avoided.

A little web security primer before we start – authentication and authorization
There usually is a confusion regarding the distinction between authorization and authentication. And of course, the fact the abbreviation *auth* is often used for both helps aggravate this common confusion. This confusion is so common that maybe this issue should be included in this post as "Common Web Vulnerability Zero".

So before we proceed, let's clearly the distinction between these two terms:

- **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.

1. **Common 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 – we'll talk about this later), 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 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 environment. 99% of the time it does not.

2. **Common 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:

1. The URL might contain the session id and leak it in the referer header to someone else.
2. The passwords might not be encrypted either in storage or transit.
3. The session ids might be predictable, thus gaining access is trivial.
4. Session fixation might be possible.
5. Session hijacking might be possible, timeouts not implemented right or using HTTP (no SSL), 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.

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

This is a fairly widespread input sanitization failure (essentially a special case of common mistake #1). 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, 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.

4. **Common 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.

5. **Common 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:

   1. Running the application with debug enabled in production.
   2. Having directory listing enabled on the server, which leaks valuable information.
   3. Running outdated software (think WordPress plugins, old PhpMyAdmin).
   4. Having unnecessary services running on the machine.
   5. Not changing default keys and passwords. (Happens way more frequently than you'd believe!)
   6. 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.

6. **Common 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, use AES (256 bits and up) and RSA (2048 bits and up).

   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 with a proper certificate and PFS (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. Do not store credit card information *ever*, as you probably don't want to have to deal with being PCI compliant. Sign up with a payment processor such as Stripe or Braintree. 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 is recommended. If you don't use bcrypt, educate yourself on salting and rainbow tables.

   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!

7. **Common 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.

8. **Common Mistake #8: Cross Site Request Forgery (CSRF)**
   This is a nice example of a confused deputy 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=46
73243243" 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 (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).

9. **Common 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 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 (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).
- *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.

10. Common 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 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.

T6.3) Challenges of web security

Overreliance on Web gateways is putting data, users, customers, organizations, and reputation in harm's way.

Once upon a time, organizations primarily used Web gateways to prevent employees from wasting time surfing the Web — or worse, from visiting gambling, adult, and other unauthorized websites.

A few decades later, Web gateways do much more than enforce regulatory compliance and HR policies. Organizations rely on them to thwart Internet-borne threats in three ways:

- Advanced URL filtering, which uses categorization, reputation analysis, and/or blacklists to control access to categories of malicious or suspicious websites.
- Anti-malware protection, which uses various capabilities (such as antivirus, sandboxing, advanced threat protection, content inspection, etc.), to guard against infections caused by various kinds of malware (including rootkits, worms, Trojans, viruses, ransomware, spyware, adware, etc.).
- Application control capabilities, which manage and limit what users are allowed to do in specific applications.

However, although Web gateways have been around for decades and continue to evolve, they aren't bulletproof, and overreliance on them is putting data, users, customers, organizations, and reputation in harm's way. Here are five of the biggest Web gatway security challenges:

1. Filtering out malicious sites
Although URL categorization sounds appealing, this approach is actually very limited. To categorize malicious sites with 100% accuracy, Web gateways need to know how to identify even the most advanced threats. Unfortunately, the attackers' rate of innovation combined with frequent zero-day exploits are leaving Web gateways behind the curve.

To make things worse, it's also hard to keep up when 571 new websites are created every second, which generates a high volume of domains and increases the chance that some will be missed by security controls. It’s difficult for filters to detect the malicious URLs that attackers use for three reasons: URLs may be triggered only by the target organization and remain stealthy during categorization, they’re short lived (less than 24 hours), and they use dynamic domains that are harder to thwart than static ones.

2. Protecting against uncategorized websites without compromising productivity
Employees need access to information to be productive. However, many organizations block access to uncategorized sites because of security concerns, and in the process they reduce end user productivity. Not only does this practice hinder end users, but security teams are forced to deal with an onslaught of support tickets for users who legitimately need to access information. As a result, security teams find themselves maintaining a growing number of policies and rules. This is a major Web security problem because 1% to 10% of URLs can't be classified because of a lack of information.

3. Fighting infections from websites considered safe
The belief that infections occur only through websites that are categorized as suspicious or malicious is false. Websense estimates that 85% of infections occur through websites considered legitimate and safe. It's becoming increasingly common for so-called safe websites to knowingly serve malicious content.

A good example is "malvertizing," which injects malicious ads into legitimate online advertising networks later served by publishers that don't know that ads are malicious. These malicious ads may not even require any user interaction to infect unsuspecting victims. A recent example is the large-scale malvertising attacks that occurred in June and July this year against several Yahoo properties. To circumvent ad blockers' ability to separate banner and display ads, some publishers are integrating ads into their general content. Others, including *GQ* publisher Condé Nast, insist that users disable their ad blockers in order to access content.

Then there's the fact that many seemingly safe websites use common content management systems that are vulnerable to zero-day exploits and can therefore be compromised by attackers to serve malicious content. In July, thousands of websites running WordPress and Joomla — which account for about 60% of all website traffic — served ransomware to all their visitors. And you may remember that back in early 2015, Forbes.com was breached by Chinese hackers who served malicious code via its "Thought of the Day" Flash widget.

4. Identifying malicious files and keeping them out
Although some Web gateways integrate antivirus engines and other file-scanning services, antivirus scanners detect only 20% to 30% of malware.

Leveraging sandboxes to detect malware requires time to run and analyze files. To avoid affecting user experience, Web gateways often pass files to users while sandboxes complete their analysis in the background — which essentially means users are exposed to attacks. Moreover, with the proliferation of sandbox evasion techniques and as malware is often target-specific, sandboxes are proving to be less effective.

5. Neutralizing malware on infected machines
Web gateways only analyze network traffic, not what users are actually doing. As such, gateways have a hard time differentiating between legitimate and malicious traffic, and detecting and neutralizing malware on infected machines. In fact, some advanced threats can be active for weeks or even months without being detected.

Indeed, recent research has found that 80% of Web gateways failed to block malicious outbound traffic. Remote access Trojans are another example of how Web gateways can't detect and stop malicious traffic.

T6.4) Explaining web security measures

A. Basic System Security Measures

The *Basic System Security Measures* apply to all systems at NYU, regardless of the level of their *System Classification*. It is a baseline, which all systems must meet. Note that for most personal workstations, these are the only Measures that apply. The requirements are:

1. **Password Protection**: All accounts and resources must be protected by passwords which meet the following requirements, which must be automatically enforced by the system:
    1. Must be at least eight characters long
    2. Must NOT be dictionary or common slang words in any language, or be readily guessable
    3. Must include at least three of the following four characteristics in any order:upper case letters, lower case letters, numbers, and special characters, such as*!@#$%^&*.
    4. Must be changed at least once per year.
2. **Software Updates**: Systems must be configured to automatically update operating system software, server applications (webserver, mailserver, database server, etc), client software (web-browsers, mail-clients, office suites, etc), and malware protection software (anti-virus, anti-spyware, etc). For *Medium* or *High Availability* systems, a plan to manually apply new updates within a documented time period is an acceptable alternative.
3. **Firewall**: Systems must be protected by a firewall which allows only those incoming connections necessary to fulfill the business need of that system. Client systems which have no business need to provide network services must deny all incoming connections. Systems that provide network services must limit access those services to the smallest reasonably manageable group of hosts that need to reach them.
4. **Malware Protection**: Systems running Microsoft or Apple operating systems must have anti-virus software installed and it must be configured to automatically scan and update.

B. Intermediate System Security Measures

The *Intermediate System Security Measures* define the Security Measures that must be applied to *medium criticality* and *high criticality* systems. Note that except under special circumstances, they do not apply to desktop and laptop computers. The requirements are:

1. **Authentication and Authorization**
    1. **Remove or disable accounts upon loss of eligibility**: Accounts which are no longer needed must be disabled in a timely fashion using an automated or documented procedure.
    2. **Separate user and administrator accounts**: Administrator accounts must not be used for non-administrative purposes. System administrators must be provisioned with non-administrator accounts for end-user activities, and a separate administrator account that is used only for system-administration purposes.
    3. **Use unique passwords for administrator accounts**: Privileged accounts must use unique passwords that are not shared among multiple systems.

Credentials which are managed centrally, such as the NetID/password combination, are considered a single account, regardless of how many systems they provide access to.

4. **Throttle repeated unsuccessful login-attempts**: A maximum rate for unsuccessful login attempts must be enforced. Account lockout is not required, but the rate of unsuccessful logins must be limited.
5. **Enable session timeout**: Sessions must be locked or closed after some reasonable period.
6. **Enforce least privilege**: Non-administrative accounts must be used whenever possible. User accounts and server processes must be granted the least-possible level of privilege that allows them to perform their function.

2. **Audit and Accountability**
    1. **Synchronize system clock**: The system clock must be synchronized to an authoritative time server run by NYU (currently tick.nyu.edu and tock.nyu.edu) at least once per day.
    2. **Enable system logging and auditing**: The facilities required to automatically generate, retain, and expire system logs must be enabled.
    3. **Follow an appropriate log retention schedule**: System logs must be retained for 30-90 days and then destroyed unless further retention is necessary due to legal, regulatory, or contractual requirements.
    4. **Audit successful logins**: Generate a log message whenever a user successfully logs on.
    5. **Audit failed login attempts**: Generate a log message whenever a user attempts to log on without success.
    6. **Audit when a system service is started or stopped**: Generate a log message when a system service is started or stopped.
    7. **Audit serious or unusual errors**: Generate a log message when a serious or unusual error occurs, such as crashes.
    8. **Audit resource exhaustion errors**: Generate a log message when a resource exhaustion error occurs, such as an out-of-memory error or an out-of-disk error.
    9. **Audit failed access attempts**: Generate a log message when an attempt to access a file or resource is denied due to insufficient privilege.
    10. **Audit permissions changes**: Generate a log message when the permissions of a user or group are changed.
    11. **Include appropriate correlation data in audit events**: For each audit event logged be sure to include sufficient information to investigate the event, including related IP address, timestamp, hostname, username, application name and/or other details as appropriate.
3. **Configuration and Maintenance**
    1. **Security Partitioning**: Systems may share hardware and resources only with other systems that have similar security requirements, regardless of their *criticality* classification. Systems which share similar security requirements have user communities of similar size and character, similar firewall profiles, and similar technical requirements. For example:

        1. Multiple systems of the same *criticality* may be aggregated together to share hardware and resources provided they have similar security requirements.
        2. *Medium criticality* systems may share hardware and resources with *low criticality* systems provided that all systems meet the *intermediate systems Security Measures*, and share similar security requirements.
    2. **Follow vendor hardening guidelines**: This document cannot be comprehensive for all systems available. Follow basic vendor recommendations to harden and secure systems.
    3. **Disable vendor default accounts and passwords**: Many systems come with default accounts which are publicly known. These accounts should be disabled.
    4. **Disable all unnecessary network services**: Processes and services which are not necessary to complete the function of a system must be disabled.
4. **Additional Requirements**
    1. **Report potential security incidents**: Potential security incidents must be reported to the IT Office of Information Security.
    2. **Security review**: During the design of the technical architecture, a review of the system must be requested from the NYU IT Office of Information Security.
    3. **Vulnerability assessment**: Before system deployment, a vulnerability assessment must be requested from the NYU IT Office of Information Security.
    4. **Physical access**: The system must reside in a locked facility, to which only authorized personnel have access.
    5. **Documentation**: Create and maintain documentation summarizing the business-process, major system components, and network communications associated with a system.

C. Advanced System Security Measures

The A*dvanced System Security Measures* define the Security Measures that must be applied to *high criticality* systems. The requirements are:

1. **Audit and Accountability**
    1. **Enable process auditing or accounting**: Enable process auditing or accounting, which generates logs information about the creation of new processes and their system activity.
    2. **Audit privilege escalation or change in privilege**: Generate a log message whenever a user changes their level of privilege.
    3. **Audit firewall denial**: Generate a log message when the host-based firewall denies a network connection.
    4. **Audit all significant application events**: Log all significant application events.
    5. **Write audit events to a separate system**: System logs must be written to a remote system in such a way that they cannot be altered by any user on the system being logged.
2. **Configuration and Maintenance**

1. **Follow advanced vendor security recommendations**: This document cannot be comprehensive for all systems and applications available. Conform to best practices and recommendations outlined in vendor security whitepapers and documentation.
2. **Host-based and network-based firewalls**: Systems must be protected by both a host-based and a network-based firewall that allows only those incoming connections necessary to fulfill the business need of that system.
3. **Configuration management process**: Configuration changes must be regulated by a documented configuration and change management process.
4. **Partitioning**: Systems may share hardware and resources only with other systems that have similar security requirements, regardless of their *criticality* classification. Systems which share similar security requirements have user communities of similar size and character, similar firewall profiles, and similar technical requirements. For example:
    1. Multiple systems of the same *criticality* may be aggregated together to share hardware and resources provided they have similar security requirements.
    2. *High criticality* systems may share hardware and resources with *medium* and *low criticality* systems provided that all systems meet the *advanced systems Security Measures*, and share similar security requirements.

3. **Additional Requirements**
    1. **Physical access**: The system must reside in a secured, managed data-center.

D. Data Handling Security Measures

These *Data Security Measures* define the minimum security requirements that must be applied to the data types defined in the *Reference for Data and System Classification*. Some data elements, such as credit card numbers and patient health records, have additional security requirements defined in external standards. In addition, access and use of University Data is covered by the *Administrative Data Management Policy*. Please be sure to consult all appropriate documents when determining the appropriate measure to safeguard your data.

The best way to safeguard sensitive data is not to handle it at all, and business processes that can be amended to reduce or eliminate dependence on *restricted data* should be corrected. For example, the University ID number can often be substituted for a social security number and poses much less risk if accidentally disclosed.

1. **Requirements for Handling Confidential Data**
    1. **Access control**: Access to *confidential data* must be provided on a least-privilege basis. No person or system should be given access to the data unless required by business process. In such cases where access is required, permission to use the data must be granted by the *data steward*.
    2. **Sharing**: *Confidential data* may be shared among the NYU community. It may be released publicly only according to well-defined business processes, and with the permission of the *data steward*.
    3. **Retention**: *Confidential data* should only be stored for as long as is necessary to accomplish the documented business process.

2. **Requirements for Handling Protected Data**
    1. **Access control**: Access to *protected data* must be provided on a least-privilege basis. No person or system should be given access to the data unless required by business process. In such cases where access is required, permission to use the data must be granted by the *data steward*.
    2. **Sharing**: *Protected data* may be shared among University employees according to well-defined business process approved by the data steward. It may be released publicly only according to well-defined business processes, and with the permission of the *data steward*.
    3. **Retention**: *Protected data* should only be stored for as long as is necessary to accomplish the documented business process.
    4. **Incident Notification**: If there is a potential security incident that may place protected data at risk of unauthorized access, the NYU IT Office of Information Security must be notified.
3. **Requirements for Handling Restricted Data**
    1. **Collection**: Restricted data should only be collected when all of the following conditions are met:
        1. The data is not available from another authoritative source, and
        2. The data is required by business process, and
        3. You have permission to collect the data from the appropriate *data steward*; or
        4. If the data is requested by the Office of General Counsel in response to litigation.
    2. **Access control**: Individuals must be granted access to restricted data on a least-privilege basis. No person or system may access the data unless required by a documented business process. In such cases where access is required, permission to use the data must be granted by the *data steward*.
    3. **Access auditing**: Enable file access auditing to log access to files containing restricted data.
    4. **Labeling**: Portable media containing restricted data should be clearly marked.
    5. **Sharing**: Access to restricted data can be granted only by a *data steward*. No individual may share restricted data with another individual who has not been granted access by a data steward.
    6. **Idle Access**: Devices which can be used to access *restricted data* must automatically lock after some period of inactivity, through the use of screensaver passwords, automatic logout, or similar controls.
    7. **Transit encryption**: Restricted data must be encrypted during transmission with a method that meets the following requirements.
        1. Cryptographic algorithm(s), the list of approved security functions.
        2. Cryptographic key lengths meet best-practices for length, given current computer processing capabilities.
        3. Both the source and destination of the transmission must be verified.
    8. **Storage encryption**: Restricted data must be encrypted using strong, public cryptographic algorithms and reasonable key lengths given current computer processing capabilities. Keys must be stored securely, and access to them provided on a least-privilege basis (see ISO 11568 for recommendations on

securing keys). If one-way hashing is used in lieu of reversible encryption, salted hashes must be used.

    1. Encrypt files containing restricted data using different keys or passwords than those used for system logon.
    2. Encrypt data stored in databases at the column-level.
    3. In addition to file and/or database encryption, implement full-disk encryption on portable devices containing restricted data.

9. **Retention**: Restricted data should only be stored for as long as is necessary to accomplish the documented business process.
10. **Destruction**: When restricted data is no longer needed it should be destroyed in accordance with applicable policies, using methods that are resistant to data-recovery attempts such as cryptographic data destruction utilities, on-site physical device destruction, or NAID certified data destruction service.