Vorlesung Nr. 10

Kryptologie II - Datum: 05.11.2018

- Zufallsgeneratoren (Teil 2)
- Kodierung und Blockchiffren (Teil 1)

Buch der Woche

Titel: Cryptography Engineering

Autor(en): Niels Ferguson, Bruce Schneier, Tadayoshi Kohno

Verlag: Wiley

Umfang: ca. 350 Seiten

Hinweise zum Inhalt:

Das Buch wurde von ausgewiesenen Krypto-Experten geschrieben, die nicht nur die technischen Grundlagen vermitteln können, sondern zugleich auch ihre Erfahrungen in der praktischen Umsetzung einfliessen lassen. So finden sich zwar keine Code-Beispiele, aber gelegentliche Warnungen hinsichtlich häufig gemachter Anwendungsfehler. Folgerichtig schließt as Buch ab mit einem Kapitel namens “The Dream of PKI”, in dem die Umsetzungshürden bei der Implementierung einer Public Key Infrastruktur beschrieben werden. Insgesamt ein Buch, das in keiner Krypto-Library fehlen sollte. Gut verständlich geschrieben, wenig Mathematik und Formelbalast, und dennoch nicht zu oberflächlich.

Anmerkung: die erste Auflage des Buches hieß “Practical Cryptography”.

Umfrage: Spannende Projektgruppen

Es ist geplant, zu einem oder mehreren Security-Themen eine freiwillige Projektgruppe ins Leben zu rufen, analog zur bereits bestehenden Hacking AG.

Es stehen unter anderem folgende Themen zur Diskussion:

- Password Cracking AG: Erstellung von Toolsets für das Cracking von Passwörtern auf unterschiedlichen Endgeräten und unterschiedlichen Anwendungen; eigene Tests, Demos, Implementierungen, etc.
- Cyber Defense AG: System Hardening, Perimeter Defense, Protection Tools, Log Analysis, etc.; Aufbau einer Testnetz-Umgebung, Verteidigung gegen simulierte Angriffe
- Big Brother AG: Fallstudien: was ist machbar?! Personenüberwachung mit Sensoren, Kameras, Tracking Apps, Wireless Tools, etc.
- Security Consulting AG: Durchführung kleiner Beratungsprojekte für/mit Unternehmen aus der Region, mit Coaching durch Dozenten
- Vortragsreihe Unternehmenssicherheit in der Praxis: Mitarbeiter aus der Sicherheitsabteilung einladen, Beispiele berichten lassen
- Vortragsreihe potentieller Arbeitgeber im Bereich Cyber Security: wer bietet welche spannenden Aufgaben und Jobs?
- Frage in die Runde: weitere Themenvorschläge?

Zufallszahlengeneratoren Fortsetzung...

Auswahl zufälliger Zahlen (1)

Eine häufige Aufgabe besteht in der Auswahl einer zufälligen Zahl in einem Intervall zwischen 0 und n.

Unser Zufallsgenerator liefert üblicherweise einen Bitstring. Ist unsere obere Schranke n eine Zweierpotenz $n = 2^k$, so wählen wir einfach einen Teilstring der Länge k und verwerfen die restlichen Ausgabebits.

Jeder der so erhaltenen 2^k Outputs entspricht dann einer Zahl zwischen 0 und n. Bei einem brauchbaren PRNG sollte dann jeder Output gleich wahrscheinlich sein.

Angenommen, unsere obere Schranke n ist keine Zweierpotenz. Wie sollen wir in diesem Fall vorgehen?

Idee: wir könnten einen (Zufalls-)Bitstring nehmen der Länge k, so dass $2^k > n$. Dann erhalten wir Zufallszahlen zwischen 0 und 2^k und reduzieren diese enfach mod n.

Auswahl zufälliger Zahlen (2)

Beispiel: n = 5, $2^k = 8$. Dann erhalten wir gleichverteilte Werte zwischen 0 und 7. Diese reduzieren wir mod 5:

0	1	2	3	4	5	6	7
0	1	2	3	4	0	1	2

Wie wir sehen, ist das modulo 5 reduzierte Ergebnis nicht mehr gleichverteilt.

Alternative: wir wählen das kleinstmögliche k, so dass $2^k \geq n$.

Liefert unser PRNG Bitstrings mit einer Länge > k, so ignorieren wir einfach die überflüssigen Bits und betrachten nur einen Teilstring der Länge k.

Auf diese Weise erhalten wir wieder gleichverteilte Werte zwischen 0 und 2^k.

Bekommen wir einen Zufallswert > n, so verwerfen wir diesen und wiederholen die Anfrage an den PRNG, bis dieser einen Wert liefert zwischen 0 und n. Diesen Wert nehmen wir als Zufallszahl.

Anmerkung: auf diese Weise "vergeuden" wir zwar einige PRNG Outputs, doch das Verfahren garantiert, dass unser Auswahlalgorithmus nicht manche Werte häufiger erzeugt als andere.

Risiken bei Entropy Pools

- Gute, “echte” Zufallswerte zum Seeding des Entropy Pools werden je nach Systemumgebung nur langsam erzeugt. Angenommen, wir haben bis zum Start unseres RNG nur etwa 40 Bit im Entropy Pool angesammelt.
- Mit heutigen Rechengeschwindigkeiten wäre es kein Problem, alle 2^{40} Initialzustände des (deterministischen) Zufallszahlengenerators durchzuprobieren. Erhalten wir zudem einen Output unseres Zufallsgenerators, so können wir prüfen, welcher der 2^{40} Initialzustände zum gegebenen Output führt. Der Zufallsgenerator wäre in diesem Fall geknackt.
- Dieses Problem lässt sich entschärfen, indem man stets darauf achtet, einen hinreichend großen Entropy Pool zu implementieren, mit weit mehr als 40 Bit Inhalt, beispielsweise 256 Bit oder mehr.
- Wenn wir dann 40 “neue” echte Zufallsbits aus diversen Entropiequellen erhalten, so mischen wir diese in den vorhandenen Poolinhalt und erhalten ein weitaus robusteres Seeding. Wichtig beim Einmischen ist die Unvorhersehbarkeit des Einmischprozesses (diverse Hashoperationen, Reihenfolge der abgefragten Entropiequellen, und vieles mehr). Exhaustive Search kann auf diese Weise verhindert werden.

Lineare Kongruenzgeneratoren (1)
Linear Congruential Generators

- Eine weit verbreitete Methode zur Erzeugung von Pseudozufallszahlen ist die Verwendung sogenannter ***Linearer Kongruenzgeneratoren***. Die wichtigste Information vorab: für kryptographische Zwecke sind diese ungeeignet.
- Warum betrachten wir diese dann überhaupt? Kryptographen möchten nicht nur sichere Verfahren konstruieren und einsetzen, sondern auch unsichere Verfahren identifizieren und deren Schwächen ausnutzen.
- Wie funktionieren lineare Kongruenzgeneratoren? Hinter dem Begriff verbirgt sich nichts anderes als eine affin lineare Abbildung. Mit der Bezeichnung “linear” ist zugleich auch schon das Problem genannt. Lineare Zusammenhänge möchte man in der Kryptographie meist vermeiden, um die Rekonstruktion von Input Parametern mittels Lösen von Gleichungssystemen zu verhindern.

Lineare Kongruenzgeneratoren (2)

Die Folge von Pseudozufallszahlen $x_0, x_1, x_2, \ldots$ ist wie folgt definiert:

$x_{n+1} = (a \cdot x_n + c) \bmod m$

Hierbei sind $0 < a < m$, $0 \leq c < m$, $0 \leq x_0 < m$.

Die Parameter a, c, x_0 und m gilt es hierbei so zu wählen, dass die Folge der x_i unseren Anforderungen an (Pseudo-)Zufallszahlen gerecht wird. Diese Anforderungen beinhalten folgendes:

- Die Folge $x_0, x_1, x_2, \ldots$ sollte möglichst lang sein, ohne dass ein Element sich wiederholt, da sich ab diesem Zeitpunkt die gesamte Teilfolge zwischen den identischen Folgegliedern stets wiederholt. Im Idealfall wünscht man die maximal mögliche Periode von m-1.
- Die Folge soll “zufällig” erscheinen.

Lineare Kongruenzgeneratoren (3)

Die Folge von Pseudozufallszahlen $x_0, x_1, x_2, \ldots$ ist wie folgt definiert:

$x_{n+1} = (a \cdot x_n + c) \bmod m$

Anmerkung 1: benötigen wir Zufallswerte zwischen Null und Eins, so dividieren wir den Output einfach stets durch m: $\frac{x_0}{m}, \frac{x_1}{m}, \frac{x_2}{m}, \ldots$

Anmerkung 2: zum Erreichen der maximalen Periode m-1 müssen a, c und m mehrere Teilbarkeitsanforderungen erfüllen. Die Anforderungen und der zugehörige Beweis sind nicht kurz, weshalb wir diese hier nicht weiter betrachten (siehe hierzu z.B. Knuth, The Art of Computer Programming Vol. II).

Lineare Kongruenzgeneratoren (4)

Zur Berechnung des n+1 -ten Folgengliedes x_{n+1} müssen wir nicht das vorherige Element x_n kennen und die Rechenvorschrift $x_{n+1} = (a \cdot x_n + c) \bmod m$ anwenden.

Stattdessen genügt es, wenn wir den Startwert x_0 kennen, dann errechnet sich x_n wie folgt:

$$x_n = \left(a^n \cdot x_0 + c \cdot \frac{a^n - 1}{a - 1}\right) \bmod m$$

Beweis: wir verwenden vollständige Induktion.

Für n = 1 gilt die Behauptung offensichtlich:

$$x_1 = \left(a^1 \cdot x_0 + c \cdot \frac{a^1 - 1}{a - 1}\right) \bmod m = (a \cdot x_0 + c) \bmod m$$

Als nächstes betrachten wir den Induktionsschritt $n \rightarrow n+1$:

Lineare Kongruenzgeneratoren (5)

$$x_{n+1} = (a \cdot x_n + c) \bmod m = a\left(a^n \cdot x_0 + c \cdot \frac{a^n - 1}{a - 1}\right) + c \bmod m =$$

$$= a^{n+1} \cdot x_0 + a \cdot c \cdot \frac{a^n - 1}{a - 1} + c \bmod m =$$

$$= a^{n+1} \cdot x_0 + c \cdot \frac{a(an - 1)}{a - 1} + c \bmod m =$$

$$= a^{n+1} \cdot x_0 + c \cdot \frac{(an^{+1} - a)}{a - 1} + c \bmod m =$$

$$= a^{n+1} \cdot x_0 + c \cdot \frac{(an^{+1} - 1 + 1 - a)}{a - 1} + c \bmod m =$$

$$= a^{n+1} \cdot x_0 + c \cdot \frac{(an^{+1} - 1) + 1 - a)}{a - 1} + c \bmod m =$$

$$= a^{n+1} \cdot x_0 + c \cdot \left(\frac{(an^{+1} - 1)}{a - 1} - 1\right) + c \bmod m =$$

$$= a^{n+1} \cdot x_0 + c \cdot \frac{(an^{+1} - 1)}{a - 1} - c + c \bmod m =$$

$$= a^{n+1} \cdot x_0 + c \cdot \frac{(an^{+1} - 1)}{a - 1} \bmod m$$

Mersenne Twister (1)

Der Pseudozufallszahlengenerator ***Mersenne Twister*** (oft mit ***MT*** abgekürzt) wurde in einer früheren Vorlesung mit einer Folie bereits erwähnt. In diesem Zusammenhang haben wir festgestellt, dass der Mersenne Twister zwar hervorragende statistische Eigenschaften aufweist, die für viele Praxisanwendungen nützlich sind.

Für kryptographische Zwecke jedoch ist MT völlig unbrauchbar. Woran liegt dies?

Die detaillierte Beschreibung des MT würde mehrere Seiten in Anspruch nehmen. Wir gehen stattdessen nur auf das Kernproblem ein, weshalb MT und verwandte Algorithmen leicht zu brechen sind.

Mersenne Twister ist ein weiteres Beispiel für ein (lineares) Verfahren, das mit linearen Gleichungssystemen geknackt werden kann.

Mersenne Twister (2)

Der MT Algorithmus verwendet ein internes Array aus 624 Bitstrings (“Worten”) der Länge 32 Bit. Mit diesem Array aus insgesamt 624·32 = 19968 Bit = ca. 2.5 kB lässt sich der gesamte interne Status dieses Zufallsgenerators beschreiben.

Zu Beginn besteht dieser Internal state aus den Worten

S_0, S_1, S_2, ..., S_{624}. Nach einer Iteration wechselt der Internal state zu

S_1, S_2, S_3, ..., S_{625}. In der nächsten Iteration dann S_2,..., S_{626}, und so weiter.

Die Outputs S_{625}, S_{626} etc. werden gebildet als Funktion vorheriger S_i. Die verwendete Funktion bildet mittels XOR Kombinationen vorheriger Worte und zweier Konstanten.

Ohne Betrachtung der Details lässt sich zu MT feststellen, dass dessen zukünftige Outputs bzw. Zustände, ebenso wie frühere Outputs bzw. Zustände, vollständig berechnet werden können, sobald wir ein Array aus mindestens 624 Worten kennen, beispielsweise, indem wir entsprechend viele Pseudozufallszahlen aus MT abrufen.

Mac OS X, iOS und PRNG

- Apple's Mac OS X verwendet für die Erzeugung von Pseudozufallszahlen den Algorithmus Yarrow, zusammen mit SHA-1.
- Mac OS X kennt ebenfalls die beiden Dateien /dev/random und /dev/urandom.
- Es macht keinen Unterschied, welche der beiden Dateien man verwendet, beide verhalten sich identisch.
- Apple's iOS verwendet ebenfalls Yarrow.

Linux Cygwin und PRNG

- Die Emulatorsoftware Cygwin, mit der man auf einem Windows Host Linux-Software laufen lassen kann, stellt ebenfalls die beiden Varianten /dev/random und /dev/urandom zur Verfügung.
- In Blogs zum Thema findet man die folgende Info (nicht verifiziert):
  - /dev/random greift zu auf die Windows-Funktion CryptGenRandom
  - /dev/urandom nutzt einen kryptographisch unsicheren linearen Kongruenzgenerator

Open BSD und PRNG

- OpenBSD verwendet /dev/random und /dev/arandom.
- Seit OpenBSD 5.5 wurde der zugrunde liegende Algorithmus von RC4 umgestellt auf ChaCha20.
- Hardware-basierte Zufallsgeneratoren werden – sofern vorhanden – vom System unterstützt. Für Details siehe OpenBSD Cryptographic Framework.

FreeBSD und PRNG

- FreeBSD kennt ebenfalls /dev/random und /dev/urandom.
- /dev/urandom ist ein Link auf /dev/random.
- Als Pseudozufallsgenerator (Algorithmus) wird Fortuna verwendet.
- Eine Entropie-Schätzung für den Entropy Pool findet nicht statt.
- Ein Blocking erfolgt nur dann, falls zu Beginn noch nicht ausreichend geseeded wurde.

Linux Kernel Entropy Pool

Wir haben bereits einige Eigenschaften von Linux sowie von /dev/random und /dev/urandom in Linux betrachtet. Wir kommen an dieser Stelle nochmals zurück und beleuchten einige weitere Details.

Wo versteckt sich der bereits erwähnte Entropy Pool und wie erhalten wir eine Information über dessen "Entropy-Zustand" (amount of entropy)? Der zugehörige Wert ist enthalten in

/proc/sys/kernel/random

Eine Abfrage ist beispielsweise möglich mittels

```
cat /proc/sys/kernel/random/entropy_avail
```

Die hierdurch ausgegebene Information ist allerdings inhaltlich kaum nachvollziehbar...

Eine Entnahme aus /dev/urandom in Form "lesbarer" Zeichen (hier: Base 64) kann beispielsweise wie folgt geschehen:

```
head –c 5 /dev/urandom | base64
```

Encoding (Kodierung, nicht Verschlüsselung)

Welche Blocklänge?

Frage in die Runde:

Welche Blocklänge empfiehlt sich für eine moderne Blockchiffre?

Gibt es je nach Blocklänge unterschiedliche Vor- und Nachteile? Warum?

Warum nicht viel kürzer?

Warum nicht viel länger?

Blocklänge und Sicherheit

Die typische Blocklänge symmetrischer Blockchiffren hat sich im Lauf der Jahre erhöht. Während DES noch auf 64-Bit Blocks operierte, haben AES und viele andere Blockchiffren eine Blocklänge von 128 Bit. Die idealen Blocklängen für eine effiziente Bearbeitung sind abhängig von der aktuellen Hardware bzw. CPUs. Im Augenblick (2018) sind 128 Bit eine gute Größe und moderne CPUs können mehrere solcher Blöcke parallel verarbeiten (z.B. Intel's AVX Advanced Vector Extensions Family of Instructions).

Die Blocklänge hat zugleich Einfluss auf die Sicherheit der Blockchiffre: zu kurze Blöcke ermöglichen dies eine ***Codebook Attacke***. Hierzu gehen wir wie folgt vor:

Angenommen, wir haben Blöcke der Länge 16 Bit. Dann berechnen wir mithilfe einer Chosen Plaintext Attacke zu jedem möglichen Plaintext den zugehörigen Ciphertext und bilden eine Tabelle mit allen $2^{16} = 65536$ Chiffretexten.

Danach können wir zu jedem Chiffretext anhand unserer Lookup Table den zugehörigen Plaintext identifizieren.

Dies erfordert nur $2^{16} \cdot 16 = 2^{20}$ Bit Speicherplatz, das entspricht 128 Kilobyte.

Für 64-Bit Blöcke bräuchten wir $2^6 \cdot 2^{64} = 2^{70}$ Speicherbits, das sind eine Million Terabyte. Für heutige Verhältnisse ist das zu viel.

Plaintext: Was genau ist das?

In der Kryptographie sprechen wir einfach immer von Plaintext, wenn etwas verschlüsselt werden soll. Das ist legitim, denn wie eine zu verschlüsselnde Bitfolge konstruiert ist, erscheint auf den ersten Blick egal.

Für die Kryptoanalyse ist allerdings von Interesse, welche Bitstrings als Input in Frage kommen und welche nicht. Jede mögliche Einschränkung und Bitstruktur der Inputmenge liefert nützliche Zusatzinformationen.

Besonders wichtig ist die Struktur des Plaintexts beim Versuch, einen Ciphertext mit kryptoanalytischen Methoden zu entschlüsseln. Wie können wir feststellen, ob ein Entschlüsselungsversuch erfolgreich war und den ursprünglichen Plaintext zutage fördert? Erhalten wir sinnvollen Text in einer natürlichen Sprache, so ist die Entscheidung einfach. Besteht der Plaintext jedoch aus Binärdaten, so ist es für einen Computer schon deutlich schwieriger zu entscheiden, ob die "entschlüsselten" Daten tatsächlich einem gesuchten Plaintext entsprechen.

Aus diesem Grund, und insbesondere auch im Rahmen der Implementierung kryptographischer Verfahren, benötigen wir Kenntnisse über die Art der Kodierung (z.B. ASCII, MIME, BASE64, UTF-8, etc.), in der der Plaintext vorbereitet wurde.

Padding

Wenn wir Daten in Form von Bitstrings (Blocks) fester Länge kodieren, so stimmt die Länge der Nutzdaten nicht immer überein mit der zugehörigen Länge der Kodierungseinheit (z.B. Byte, Register, Cipher block, etc.).

In solchen Fällen muss der entstehende “Rest” bis zur jeweiligen Blocklänge irgendwie aufgefüllt werden. Dies nennt man ***Padding***.

Überlassen wir dem Zufall, welche Bits im Rest des Strings stehen, so kann dies zu Fehlinterpretation kommen, da nicht ersichtlich ist, wo die Nutzdaten aufhören und wo die “Füllbits” beginnen. Dies wäre beispielsweise dann der Fall, wenn restliche Bits einfach mit Nullen aufgefüllt würden, da dann nicht ersichtlich ist, welche Nullen evtl. noch Teil der Nutzdaten waren.

Wir müssen also anhand einer geeigneten Zeichenkette signalisieren, wo die Füllbits beginnen. Hierbei müssen wir darauf achten, dass eine solche Zeichenkette nicht auch Bestandteil der Nutzdaten sein könnteund als solche interpretiert wird.

In der Kryptographie wiederum ist Padding problematisch, da es zu einem Known Plaintext führen kann und die Kryptoanalyse dadurch ggf. erleichtert.

Wir werden in Kürze mehrere Beispiele für Padding genauer betrachten.

ASCII

Der ASCII Zeichensatz wurde in den 60er Jahren entwickelt und diente zu Beginn für Telegraphen-artige Geräte. Zu Beginn der Computer-Ära war ASCII der gemeinhin verwendete Zeichensatz.

ASCII umfasst 128 Zeichen und lässt sich folglich mit 7 Bit kodieren.

ASCII wird in Abgrenzung zu späteren Erweiterungen und Internationalisierungen auch als US-ASCII bezeichnet. US-ASCII enthält sowohl druckbare als auch nicht druckbare Zeichen, allerdings keine Zeichen wie ö, ä, ü, ß, @, etc.

ASCII umfasst 95 druckbare Zeichen. Die ersten 32 Zeichen der ASCII-Tabelle sind nicht druckbare (Steuer-)Zeichen. Steuerzeichen konnten beispielsweise in der Kommunikation mit Druckern eine Anweisung wie “Line Feed” enthalten, mittels derer die Druckerpapierrolle eine Zeile fortbewegt wurde.

Bis 2007 war ASCII der dominierende Zeichensatz im WWW und wurde danach weitgehend abgelöst von UTF-8. Die ersten 128 Zeichen von UTF-8 sind identisch mit ASCII, so dass Abwärtskompatibilität gegeben ist.

Internationale ASCII-Varianten tauschen einzelne Zeichen des US-ASCII durch eigene (z.B. Umlaute) aus. Dadurch fehlen andere ggf. wichtige Zeichen. Ein Programmierer, der z.B.

`{ a[i] = '\n'; }` schreiben wollte, erhält im lokalen Zeichensatz stattdessen

`ä aÄiÜ = 'Ön’; ü`.

ASCII bildet die Grundlage späterer Erweiterungen und Internationalisierungen und spielt auch heute noch eine wichtige Rolle.

ASCII Tabelle

RFC 20 ASCII format for Network Interchange October 1969

2. Standard Code

B I T S \ b7				0	0	0	0	1	1	1	1
			b6	0	0	1	1	0	0	1	1
			b5	0	1	0	1	0	1	0	1
			COLUMN->	0	1	2	3	4	5	6	7
b4	b3	b2	b1	ROW							
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

3. Character Representation and Code Identification

The standard 7-bit character representation, with b7 the high-order bit and b1 the low-order bit, is shown below:

EXAMPLE: The bit representation for the character "K," positioned in column 4, row 11, is

```
b7 b6 b5 b4 b3 b2 b1
1  0  0  1  0  1  1
```

ISO/IEC 8859

ISO/IEC 8859 ist ein Standard zur 8-Bit Zeichen-Kodierung. Es enthält die 128 ASCII-Zeichen. Weitere 128 Zeichen erlauben national gebräuchliche Buchstaben und Symbole.

ISO/IEC 8859 gibt es folglich in verschiedenen internationalisierten Ausprägungen, ISO/IEC 8859-1 bis ISO/IEC 8859-15. Die Version ***ISO/IEC 8859-1***, auch ***Latin-1*** genannt, enthält alle in Deutschland gebräuchlichen Buchstaben (Umlaute etc.) und wird hier daher oft verwendet. Sehr ähnlich zu ISO/IEC 8859-1 ist ISO/IEC 8859-15, das zusätzlich das Euro-Symbol enthält.

ISO/IEC 8859 hat eine sehr hohe Verbreitung und ist beispielsweise Standard überall dort, wo ein Dokumentenaustausch über HTML im MIME Type “text/” erfolgt. Bei Betriebssystemen und Anwendungen im Auslieferungszustand ist ISO 8859-1 oft standardmäßig voreingestellt.

In einigen Fällen wird dort, wo ein Zeichensatz als ISO/IEC8859-1 bezeichnet wird, de facto ein ähnlicher, aber hiervon abweichender Zeichensatz namens ***Windows-1252*** verwendet (so beispielsweise bei HTML 5 documents).

ISO/IEC 8859 wurde allerdings zunehmend abgelöst durch Unicode.

Nicht alle Anwendungen (insbesondere E-Mail) verstehen ISO/IEC 8859.

ISO 8859-1 (Latin 1) Tabelle

ISO/IEC 8859-1

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_																
1_																
2_	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	
8_																
9_																
A_	NBSP 00A0	¡ 00A1	¢ 00A2	£ 00A3	¤ 00A4	¥ 00A5	¦ 00A6	§ 00A7	¨ 00A8	© 00A9	ª 00AA	« 00AB	¬ 00AC	SHY 00AD	® 00AE	¯ 00AF
B_	° 00B0	± 00B1	² 00B2	³ 00B3	´ 00B4	µ 00B5	¶ 00B6	· 00B7	¸ 00B8	¹ 00B9	º 00BA	» 00BB	¼ 00BC	½ 00BD	¾ 00BE	¿ 00BF
C_	À 00C0	Á 00C1	Â 00C2	Ã 00C3	Ä 00C4	Å 00C5	Æ 00C6	Ç 00C7	È 00C8	É 00C9	Ê 00CA	Ë 00CB	Ì 00CC	Í 00CD	Î 00CE	Ï 00CF
D_	Ð 00D0	Ñ 00D1	Ò 00D2	Ó 00D3	Ô 00D4	Õ 00D5	Ö 00D6	× 00D7	Ø 00D8	Ù 00D9	Ú 00DA	Û 00DB	Ü 00DC	Ý 00DD	Þ 00DE	ß 00DF
E_	à 00E0	á 00E1	â 00E2	ã 00E3	ä 00E4	å 00E5	æ 00E6	ç 00E7	è 00E8	é 00E9	ê 00EA	ë 00EB	ì 00EC	í 00ED	î 00EE	ï 00EF
F_	ð 00F0	ñ 00F1	ò 00F2	ó 00F3	ô 00F4	õ 00F5	ö 00F6	÷ 00F7	ø 00F8	ù 00F9	ú 00FA	û 00FB	ü 00FC	ý 00FD	þ 00FE	ÿ 00FF

Letter Number Punctuation Symbol Other undefined undefined in the first release of ECMA-94 (1985).

Unicode

Unicode ist das Ergebnis einer Initiative von ISO und dem Unicode-Konsortium, alle existierenden Sprachen und Zeichensätze in einer gemeinsamen Kodierung unterzubringen.

Die zugehörige Spezifikation der Zeichensätze heißt ***Universal Coded Character Set UCS*** und findet sich in ***ISO/IEC 10646***. Unicode beinhaltet mehr als nur die Beschreibung des Zeichensatzes und ist daher nicht gleichzusetzen mit UCS.

Aktuell (2018) haben wir ISO/IEC 10646:2017 sowie Unicode 10.0.

Da hierzu mehr als 256 Zeichen erforderlich sind, muss man die Zeichen eines Zeichensatzes je nach Anforderung nicht nur mit einem, sondern mit zwei bis vier Bytes kodieren (UTF-8, UTF-16, UTF-32). Die dadurch gewonnene Flexibilität wird also erkauft durch höheren Speicherplatzbedarf.

Die ersten 256 Zeichen in Unicode sind identisch mit jenen aus ISO/IEC-8859-1.

Unicode Fonts

This is a list of Unicode fonts, including open-source Unicode typefaces, showing the number of characters/glyphs included for the released version, and also showing font's license type:

- Alphabetum (shareware, includes a few SMP character blocks. Over 5,490 characters in version 9.00)
- Arial Unicode MS (distributed along with Microsoft Office (2002XP, 2003). only supports up to Unicode 2.0. Contains 50,377 glyphs (38,917 characters) in v1.01.)
- Batang and Gungsuh, a serif and monospace/gothic font, respectively; both with 20,609 Latin/Cyrillic/CJK glyphs in version 2.11. Distributed with Microsoft Office.
- Bitstream Cyberbit (free for non-commercial use. 29,934 glyphs in v2.0-beta.)
- Bitstream Vera (free/open source, limited coverage with 300 glyphs, DejaVu fonts extend Bitstream Vera with thousands of glyphs)
- Charis SIL (free/open source, over 4,600 glyphs in v4.114)
- Code2000 (shareware Unicode font; supports the entire BMP. 63,888 glyphs in v1.15. Abandoned.)
  - Code2001 (freeware; supports the SMP. 2,944 glyphs in v0.917. Abandoned.)
  - Code2002
- DejaVu fonts (free/open source, "DejaVu Sans" includes 3,471 glyphs and 2,558 kerning pairs in v2.6)
- Doulos SIL (free/open source, designed for IPA, 3,083 glyphs in v4.014.)
- EB Garamond (free/open source, includes 3,218 glyphs in 2017)
- Everson Mono (also known as, Everson Mono Unicode. Shareware; contains all non-CJK characters. 4,899 glyphs in Macromedia Fontographer v4.1.3 2003-02-13.)
- Fallback font (freeware fallback font for Windows)
- Fixedsys Excelsior (freeware, 5,992 glyphs in v3.01, supports most glyphs in the basic plane except CJK)
- Free UCS Outline Fonts aka FreeFont (free/open source, "FreeSerif" includes 3,914 glyphs in v1.52, MES-1 compliant)
- Gentium (free/open source, "Gentium Plus" includes over 5,500 glyphs in November 2010)
- GNU Unifont (free/open source, bitmapped glyphs are inclusive as defined in unicode-5.1 only)
- Georgia Ref (also distributed under the name "MS Reference Serif," extension of the Georgia typeface)
- Gulim/New Gulim and Dotum, rounded sans-serif and non-rounded sans-serif respectively, (distributed with Microsoft Office 2000. wide range of CJK (Korean) characters. 49,284 glyphs
- Junicode (free; includes many obsolete scripts, intended for mediævalists. 2,235 glyphs in v0.6.12.)
- Kelvinch (free/open source, SIL Open Font License) approximately 3500 glyphs per style. Contains most Latin blocks, Cyrillic, Georgian and Armenian.
- LastResort (fallback font covering all 17 Unicode planes, included with Mac OS 8.5 and up)
- Lucida Grande (Unicode font included with macOS; includes 1,266 glyphs)*
- Lucida Sans Unicode (included in more recent Microsoft Windows versions; only supports ISO 8859-x characters. 1,776 glyphs in v2.00.)*
- MS Gothic (distributed with Microsoft Office, 14,965 glyphs in v2.30)
- MS Mincho (distributed with Microsoft Office, 14,965 glyphs in v2.30)
- Nimbus Sans Global
- Noto, a family of fonts designed by Google
- PragmataPro, a modular monospaced font family designed by Fabrizio Schiavi, Regular version includes more than 7000 glyphs
- Squarish Sans CT v0.10 (1,756 glyphs; Latin, Greek, Cyrillic, Hebrew, and more)
- STIX (especially mathematics, symbols and Greek, see also XITS)
- Titus Cyberbit Basic (free; updated version of Cyberbit. 9,779 glyphs in v3.0, 2000.)
- Verdana Ref (also distributed under the name "MS Reference Sans Serif," extension of the Verdana typeface)
- XITS (especially mathematics, symbols and Greek)

Quelle: Wikipedia

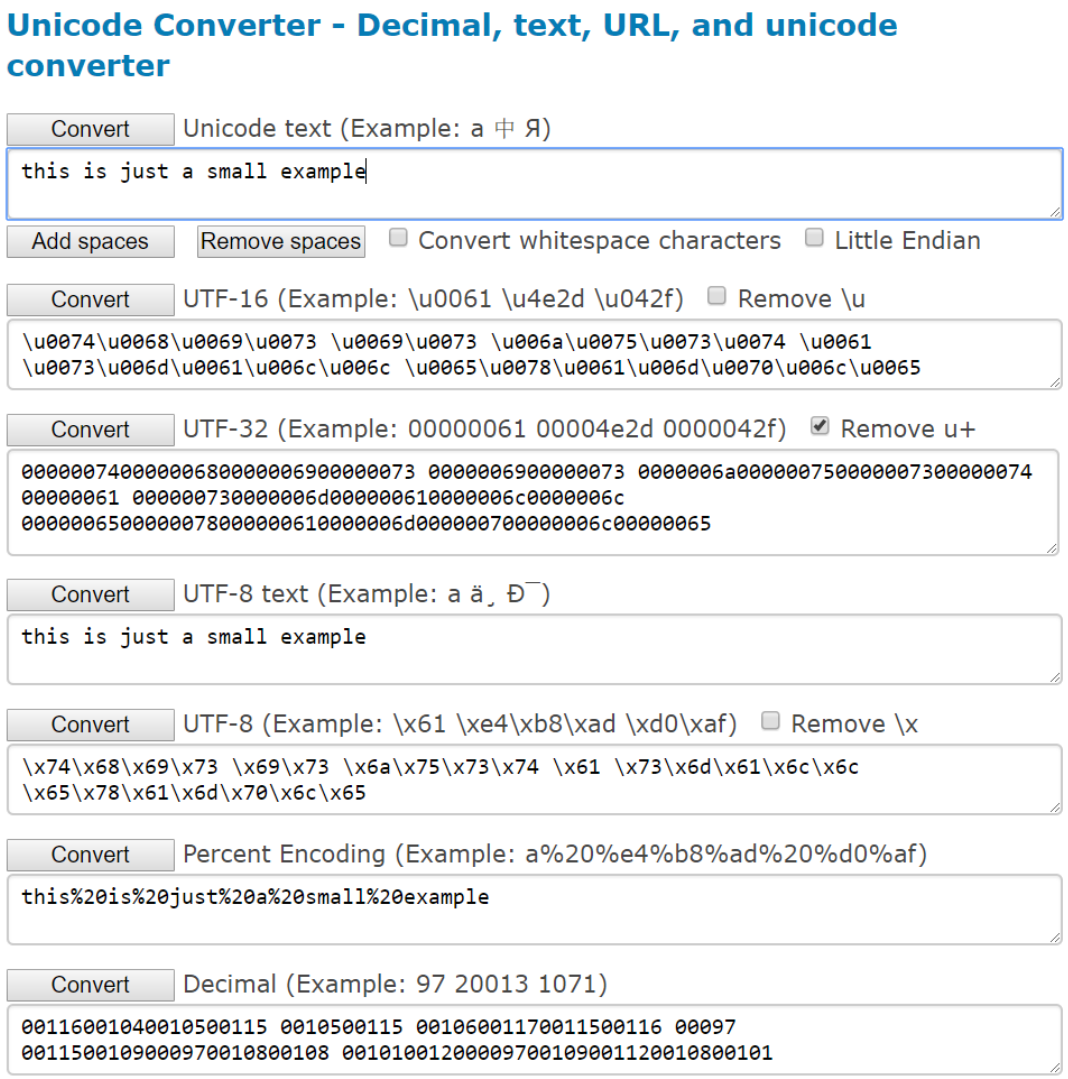
UTF-8

- UTF-8 nutzt eine variable Kodierungslänge zwischen einem und vier Byte (also 8 Bit bis 32 Bit) pro Zeichen. Hierdurch kann UTF-8 alle Unicode-Zeichen darstellen (etwas mehr als eine Million Zeichen).
- UTF-8 ist mit großem Abstand die verbreitetste Kodierung für Webseiten (92% in Sept. 2018). Das Internet Mail Consortium IMC empfiehlt, dass alle Mailprogramme Mails in UTF-8 erstellen und darstellen können. Das W3C empfiehlt UTF-8 als Standardkodierung für HTML und XML.
- Die ersten 128 Unicode-Zeichen entsprechen den 128 ASCII-Zeichen, in derselben Kodierung mit einem Byte Länge. Auf diese Weise werden ASCII-Zeichen zugleich als gültige UTF-8 Zeichen erkannt.
- Die nachfolgenden 1920 Zeichen werden mithilfe zweier Bytes dargestellt. Dies beinhaltet lateinische Zeichen, hebräische, arabische, und einige mehr. Für chinesische, japanische und koreanische Zeichen werden zumeist drei Byte benötigt. Vier Byte braucht man unter anderem für Emoji und zahlreiche mathematische Symbole.

UTF-16 und UTF-32

- UTF-16 nutzt eine variable Kodierungslänge mit ein oder zwei 16-Bit Vektoren (Words).
- UTF-16 findet Anwendung in Windows, Java, JavaScript, normalerweise aber nicht in Unix, Linux und Mac OSX. MS Windows nutzt UTF-16. Mittlerweile unterstützt Windows aber zunehmend auch UTF-8.
- Texte in europäischen Sprachen können in UTF-8 effizienter dargestellt werden als in UTF-16.
- Texte in chinesischer Sprache können in UTF-16 effizienter dargestellt werden als in UTF-8. Dies gilt aber nur dann, wenn der Anteil chinesischer Zeichen dominiert und nicht wie beispielsweise in HTML ein hoher Anteil ASCII Formatierungs-Schlüsselworte enthalten ist.
- UTF-32 nutzt stets 32 Bit zur Darstellung eines Zeichens. Der Speicherbedarf ist dementsprechend deutlich höher.
- UTF-16 und UTF-32 sind inkompatibel mit ASCII, auch dann, wenn nur ASCII-Zeichen verwendet werden.

Known Plaintext durch Kodierung



Das verwendete Format für die Kodierung des Plaintexts kann dazu führen, dass einzelne Zeichen oder Zeichenkombinationen überproportional häufig auftreten. Man beachte den Beispieltext “this is just a small example” in seiner Repräsentation als UTF (siehe rechts).

Quelle: https://www.branah.com/unicode-converter

Base64

Base64 (1)

- ***Base64*** erlaubt die Überführung binärer Daten in eine Darstellung als ASCII-Zeichen. Ziel ist die Repräsentation von Daten mithilfe solcher Zeichen(sätze), die auf den meisten Systemen vorhanden sind, auch beim Datentransfer nicht verändert werden müssen und zudem anzeigbar/druckbar sind.
- Von den 128 ASCII-Zeichen werden 64 druckbare Zeichen ausgewählt, die auf möglichst vielen Systemen international verfügbar sind. Die konkrete Auswahl ist leider nicht in jedem Fall dieselbe. Zumeist (nicht immer) verwendet Base64 die Zeichen a...z, A...Z, 0...9. Dies ergibt 62 Zeichen. Bei den verbleibenden zwei Zeichen herrscht weniger Einigkeit. Oft findet man hier “+” und “/”.
- Hinzu kommt ein 65tes Zeichen, das Gleichheitszeichen “=“ für das sogenannte Padding (mehr dazu in Kürze).
- Für die Umwandlung werden je 6 Bit Binärdaten in eines von insgesamt $2^6 = 64$ ASCII-Zeichen umgewandelt. Zur Darstellung von drei Byte Binärdaten (3x8 Bit) benötigen wir folglich vier ASCII-Zeichen. Die umgewandelte Nachricht benötigt daher ca. $\frac{4}{3}$ des vorherigen Speicherplatzes.
- Die genaue Prozedur beschreiben wir in Kürze anhand eines Beispiels.

Base64 (2)

- Viele Implementierungen beinhalten die beiden Zeichen “+” und “/”. Beispiele sind Standard Base64, ***MIME***, ***Radix-64*** (OpenPGP).
- Für ***IMAP Mailbox Names*** wird statt des “/” ein “;” verwendet.
- Für URLs und Dateinamen gibt es ***base64url***, dieses nutzt “-” statt “+” und “_” statt “/”, da letzteres z.B. in URLs fehlinterpretiert würde und in Dateinamen nicht zulässig ist.
- Eine weitere Ausnahme bildet ***uuencode***: dieses verwendet diverse Satzzeichen anstelle der Kleinbuchstaben. uuencode wurde schon früh in Unix-Systemen verwendet, da alte Drucker damals alle Buchstaben als Großbuchstaben druckten. uuencode nutzt daher die ASCII-Zeichen Nr. 32 (Space) bis Nr. 95 (“_”):

“ !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_”

Anmerkung: die Anführungszeichen am Anfang und Ende gehören nicht dazu.

Base64 und Hacking

Frage in die Runde:

spielt Base64 eine Rolle in der Cyber Security?

Wo? Warum?

Base64 (3)

- Unix/Linux speichert Password Hashes in der Datei /etc/passwd in einem Encoding namens ***B64***:

"./0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz“

- Die Hashfunktion ***bcrypt*** verwendet wieder eine andere Form, wie folgt:

"./ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789“

- Base64 Encoding wird auch von Malware und/oder Spammern gerne genutzt zur Umgehung von Filtern bzw. Input Validation, da eine Rückumwandlung von Base64 oft erst im Zielsystem erfolgt und manche Filter keine Base64 Validierung vornehmen. Unerwünschte Code- oder Textbestandteile werden so nicht erkannt.
- Webseiten verwenden als Zugriffsschutz gelegentlich Basic Authentication: hierbei werden Benutzername und Password, getrennt durch einen Doppelpunkt, aneinandergefügt und dann in Base64 umgewandelt. Dieser String wird dann vom Client an den Server übermittelt.
- E-Mail verwendet Base64 (MIME Encoding) zur Übermittlung von Binärdaten.
- Encrypted Cookies nutzen Base64
- Skripte, HTML und XML nutzen Base64, um Binärdaten einzubinden.

Base64 Tabelle

Table 1: The Base64 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

Base64 Varianten

Variant	Char for index 62	Char for index 63	*pad* char	Fixed encoded line-length	Maximum encoded line length	Line separators
Original Base64 for Privacy-Enhanced Mail (PEM) (RFC 1421, deprecated)	+	/	= *(mandatory)*	Yes (except last line)	64	CR+LF
Base64 transfer encoding for MIME (RFC 2045)	+	/	= *(mandatory)*	No (variable)	76	CR+LF
Standard 'base64' encoding for RFC 3548 or RFC 4648	+	/	= *(mandatory unless specified by referencing document)*	No *(unless specified by referencing document)*	none *(unless specified by referencing document)*	none *(unless specified by referencing document)*
'Radix-64' encoding for OpenPGP (RFC 4880)	+	/	= *(mandatory)*	No (variable)	76	CR+LF
Modified Base64 encoding for UTF-7 (RFC 1642, obsoleted)	+	/	none	No (variable)	none	none
Modified Base64 encoding for IMAP mailbox names (RFC 3501)	+	,	none	No (variable)	none	none
Standard 'base64url' with URL and Filename Safe Alphabet (RFC 4648 §5 'Table 2: The "URL and Filename safe" Base 64 Alphabet')	-	_	= *(optional)*	No (variable)	*(application-dependent)*	none
Non-standard URL-safe Modification of Base64 used in YUI Library (Y64)[5]	.	_	-	No (variable)	*(application-dependent)*	none
Modified Base64 for XML name tokens (*Nmtoken*)	.	-	none	No (variable)	*(XML parser-dependent)*	none
Modified Base64 for XML identifiers (*Name*)	_	:	none	No (variable)	*(XML parser-dependent)*	none
Modified Base64 for Program identifiers (variant 1, non standard)	_	-	none	No (variable)	*(language/system-dependent)*	none
Modified Base64 for Program identifiers (variant 2, non standard)	.	_	none	No (variable)	*(language/system-dependent)*	none
Non-standard URL-safe Modification of Base64 used in Freenet	~	-	=	No (variable)	*(application-dependent)*	none

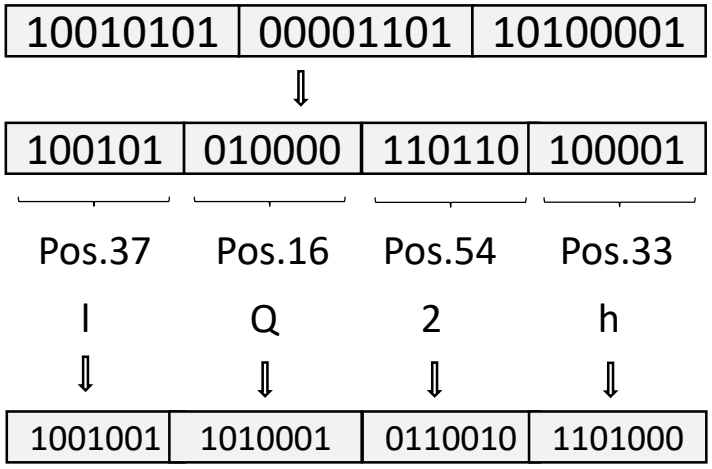
Quelle: Wikipedia

Base64 Umwandlung

- Die Umwandlung von Binärdaten in Base64 erfolgt blockweise. Der binäre Datenstrom wird hierzu in Abschnitte mit je 3 Bytes aufgeteilt (das Most significant bit eines jeden Bytes linksaußen).
- Jede Einheit mit 3 Bytes bzw. 24 Bit wird dann neu aufgeteilt in 4 Blocks der Länge 6 Bit.
- Jeder 6-Bit Vektor wird dann umgewandelt in ein Zeichen aus dem Base64 Zeichensatz.
- Da die Länge des binären Datenstroms nicht immer ein Vielfaches von 3 ist, müssen wir am Ende gegebenenfalls auf ein Vielfaches von 3 auffüllen. Dieser Prozess wird als Padding bezeichnet.
- Um Padding-Daten eindeutig von Nutzdaten zu unterscheiden, verwendet man hierbei ein 65tes Zeichen, das Gleichheitszeichen “=“.
- Die einzelnen Umwandlungsschritte betrachten wir nachfolgend anhand eines Beispiels.

Base64 Umwandlung: Beispiel

Beispiel: Wir konvertieren drei Byte Binärdaten in eine Base64 Darstellung. Das Ergebnis stellen wir dar in Form von 4 druckbaren Zeichen. Für diese 4 ASCII-Zeichen benötigen wir mehr als die ursprünglichen 3x8 = 24 Bit. Wie groß der Overhead ausfällt, hängt ab von der internen Repräsentation der druckbaren ASCII-Zeichen (standardmäßig 8 Bit pro Zeichen, ggf. auch mehr), sowie von der Anzahl erforderlicher Paddings (mehr dazu in Kürze).



← 3 Byte Binärdaten als Input

← Blocklänge wird von 8 auf 6 verändert

← Position in der Base64 Tabelle

← zugehöriges druckbares Zeichen an der Position in der Base64 Tabelle

← Binärdarstellung des druckbaren Zeichens aus der der ASCII Tabelle (128 = 2^7 Zeichen erfordern 7 Bit, de facto aber je ein Byte)

Base64 Padding

Wie wir gesehen haben, unterteilt Base64 je drei 8-Bit Vektoren in 4 6-Bit Vektoren. Was tun wir, wenn am Ende nicht drei sondern nur ein oder zwei 8-Bit Vektoren zur Umwandlung übrigbleiben? In diesem Fall füllen wir am Ende des Datenstroms so viele Bits hinzu, dass auch die letzte “Verarbeitungseinheit” aus vier kompletten 6-Bit Vektoren besteht:

a. Haben wir einen 8-Bit Vektor, so müssen wir zunächst 4 Bit (Nullen) auffüllen, um 12 Bit bzw. zwei komplette 6-Bit Vektoren für Base64 zu bekommen. Diese zwei Base64-Zeichen ergänzen wir durch zwei “=“ Zeichen.
b. Haben wir zwei 8-Bit Vektoren, so müssen wir zunächst 2 Bit (Nullen) auffüllen, um 18 Bit bzw. drei komplette 6-Bit Vektoren für Base64 zu bekommen. Diese drei Base64-Zeichen ergänzen wir durch ein “=“ Zeichen.
c. Haben wir drei 8-Bit Vektoren, so ist unser Block vollständig und wir müssen nichts tun. Da in diesem Fall nicht anhand von “=“ Zeichen das Ende des Datenstroms zu erkennen ist, muss dies im Bedarfsfall auf andere Weise ermöglicht werden.

Anmerkung: würden wir nur die 4 Bit in (a) oder die 2 Bit in (b) ergänzen, so wäre dies nicht ausreichend um zu signalisieren, ob und wo das Padding beginnt. Daher werden in beiden Fällen insgesamt vier 6-Bit Vektoren gebildet.

Anmerkung: die Art des Paddings bewirkt, dass Inputtext am Ende des Datenstroms nicht immer auf dieselben Base64 Zeichen abgebildet wird. Dies ist vielmehr abhängig von deren Position im letzten 3x8 Bit Verarbeitungsblock. Gute Beispiele finden sich auf https://en.wikipedia.org/wiki/Base64#Padding

Erkennung von BASE64 Plaintext

Frage in die Runde:

Angenommen, wir haben einen automatisierten Prozess zur Kryptoanalyse von Chiffretexten. Wir wissen zudem, dass voraussichtlich BASE64 Daten verschlüsselt wurden. Wie sehen wir dem Output eines Entschlüsselungsversuchs an, ob es sich um den Plaintext handelt, und nicht um irgendeinen Datensalat?

Angenommen es handelt sich nicht um Base64, sondern eine andere Kodierung wie beispielsweise ISO 8859 oder UTF-8. Wie könnte man dann vorgehen?

PKCS#7 Padding von Plaintext Blocks

Wenn wir blockweise verschlüsseln, können wir nicht voraussetzen, dass unser Input immer mit einer vorgegebenen Blocklänge übereinstimmt. Daher benötigen wir Padding zum Auffüllen von Blocks.

Für Blockchiffren findet man hierzu Anleitungen im ***PKCS#7*** Standard, in RFC2315 und in RFC5652 (Cryptographic Mesage Syntax).

Angenommen wir haben einen 128-Bit bzw. 16 Byte Block. Padding gemäß PKCS#7 verläuft dann folgendermaßen:

- haben wir nur ein Byte Nutzdaten und müssen noch 15 Bytes auffüllen, so nehmen wir hierfür 15x das Byte $0F_{hex}$ bzw. 15_{dec}.
- haben wir zwei Bytes Nutzdaten und müssen noch 14 Bytes auffüllen, so nehmen wir hierfür 14x das Byte $0E_{hex}$ bzw. 14_{dec}.
- ...
- haben wir 15 Bytes Nutzdaten und müssen noch ein Byte auffüllen, so nehmen wir hierfür 1x das Byte 01_{hex} bzw. 01_{dec}.
- haben wir bereits einen vollen Block mit 16 Bytes Nutzdaten und müssen eigentlich nichts mehr auffüllen, so signalisieren wir dies, indem wir einen kompletten weiteren Block anfügen und diesen anfüllen mit 16x dem Byte 10_{hex} bzw. 16_{dec}.